Universidade de Aveiro
Departamento de Eletrónica, Telecomunicações e Informática,
2015

Luís Diogo
Medina Duarte

DLL architecture for OFDM based VLC transceivers in FPGA

Desenho da camada DLL para sistemas de comunicação por luz visível

Luís Diogo Medina Duarte

DLL architecture for OFDM based VLC transceivers in FPGA

Desenho da camada DLL para sistemas de comunicação por luz visível

Dissertation project presented to the University of Aveiro to fulfil the necessary requirements for obtaining the Master degree in Electronics and Telecommunications Engineering, realized under the scientific supervision of the Doctor Luís Filipe Mesquita Nero Moreira Alves, Assistant Professor of the Electronics, Telecommunications and Informatics Department of the University of Aveiro and the Doctor António Luís Jesus Teixeira, Associate Professor with aggregation of the Electronics, Telecommunications and Informatics Department of the University of Aveiro.

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e Telecomunicações, realizada sob a orientação científica do Doutor Luís Filipe Mesquita Nero Moreira Alves (orientador), Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro e do Doutor António Luís Jesus Teixeira (coorientador), Professor Associado c/Agregação do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president

Prof. Doutor Paulo Miguel Nepomuceno Pereira Monteiro
Professor Associado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

vogais / examiners committee

Prof. Doutor Carlos Miguel Nogueira Gaspar Ribeiro
Professor Adjunto do Departamento de Engenharia Eletrotécnica do Instituto Politécnico de Leiria

Prof. Doutor Luís Filipe Mesquita Nero Moreira Alves
Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

agradecimentos

É com extrema dificuldade, devido ao curto espaço que esta secção de agradecimentos apresenta, que exprimo algumas palavras de agradecimento e profundo reconhecimento a todos os que contribuíram de forma directa e indireta para a realização desta dissertação de Mestrado.

Em primeiro lugar ao meu orientador Professor Doutor Luís Nero Alves pela oportunidade, por todo o reconhecimento, pela exigência de método e rigor, pela revisão crítica do texto, pela acessibilidade, cordialidade, simpatia e por todo o valioso apoio que me prestou para a realização deste trabalho. Ao meu coorientador Professor Doutor António Teixeira por todo o tempo e apoio disponibilizados.

À Professora Doutora Mónica Figueiredo por toda a preciosa ajuda na definição do objecto de estudo, esclarecimentos, opiniões e sugestões que se tornaram decisivos em muitos momentos da elaboração desta dissertação.

Ao meu pai por todo o apoio incondicional demonstrado, sempre do meu lado nos bons e maus momentos do meu percurso académico. Agradeço a sua compreensão inestimável, os seus diversos sacrifícios suportados e os seus exemplos dados de força e trabalho.

À minha mãe que mesmo não estando presente fisicamente, sempre me acompanhou neste percurso com o seu espírito único de uma lutadora que me fortaleceu e apoiou em todos os momentos.

Aos meus avós Idalisa, Isabel, Sílvio e Vítor por todo o apoio categórico e por todo o incentivo para alcançar caminhos cada vez mais ambiciosos.

À minha afilhada Carolina, primo Ricardo, tia Cristina e tio Ricardo por todo o carinho, pelo apoio e compreensão da importância do meu percurso académico.

À Daniela por todo o seu apoio absoluto, pela sua grande paciência e compreensão, pela confiança depositada em mim, pela valorização tão entusiasta do meu trabalho que me ajudou a superar muitos obstáculos, por todo o carinho, e acima de tudo por todo o seu amor incondicional.

Ao meu grande amigo e colega Luís por todo o apoio, todas as discussões perteninentes e pelas longas horas de trabalho partilhadas ao longo deste Mestrado. Sem ele, a escrita desta dissertação teria sido penosa. Ao meu amigo Pedro por toda a ajuda, troca de ideias e por toda a sua boa disposição que contagia o laboratório. A estes quero expressar o meu enorme agradecimento pela partilha de horas de trabalho no projecto VLCLighting.

Aos meus amigos e colegas Filipe, André, Miguel, Nuno e Gonçalo pelo bom ambiente gerado que foi propício à elaboração deste trabalho.

À minha grande amiga Ana por todo apoio e pela experiência memorável que partilhámos.

À minha amiga Joana por todo o apoio, carinho e paciência que esteve presente em tantos dias partilhados e que contribuíram como motivação para este trabalho.

Ao Instituto de Telecomunicações de Aveiro pelo espaço disponibilizado e pelo material que foi fundamental para a elaboração deste trabalho.

Resumo

Com a chegada da era da Informação, os sistemas de comunicação tornaram-se na espinha dorsal da nossa sociedade. A Sociedade Moderna esforça-se por ter acesso instantâneo a fontes de informação específicas para tomar decisões limitadas pelo tempo. Portanto, o século XXI está marcado pela crescente exigência da largura de banda nas comunicações sem fios, pois tal permite aos utilizadores comunicarem e acederem às aplicações a partir de áreas longínquas. Até ao momento, foram alcançados diversos avanços/descobertas na largura de banda das comunicações sem fios, mas tal tem sido conseguido usando o intervalo de radiofrequências (RF) do espectro eletromagnético e que fez com que o RF ficasse com o papel principal nos sistemas de comunicação de hoje. Contudo, a Tecnologia RF é vítima do seu próprio sucesso. Devido ao tremendo aumento do número de aparelhos de comunicação móveis, a tecnologia RF não pode lidar muito mais tempo com a exigência dos mercados e atingirá o seu ponto de saturação.

VLC (Comunicação através de luz visível) é uma técnica recente muito apelativa no campo das comunicações sem-fios e que pretende ser um complemento à tecnologia RF, sendo considerada por muitos investigadores como uma alternativa viável. Esta dissertação discute o problema de se alcançar uma grande taxa de transmissão com a implementação de uma Data Link Layer (DLL) direccionada para sistemas VLC com modulação OFDM e está inserida num projecto financiado intitulado VLCLighting. O objectivo principal desta dissertação consiste na implementação de um DLL eficiente num processador Microblaze numa Field-Programmable Gate Array (FPGA) e no estudo da sua utilização em sistemas VLC para uso combinado em sistemas de iluminação. Uma vez que foram identificados dois serviços com valor-acrescentado para serem incluídos no projecto VLCLighting, a proposta DLL pretende fornecer os meios necessários à fragmentação e encaminhamento das exigências dos serviços, enquanto se mantém um fluxo contínuo de transmissão capaz de assegurar as funcionalidades de iluminação e comunicação. A presente dissertação propõe um desenho inspirado nos sistemas DVB e do projeto OMEGA, e descrevendo as alterações exigidas para satisfazer os objectivos do projecto VLCLighting.

Abstract

With the advent of the Information Age, communication systems have become the backbone of our society. The modern society strives to find instant access to specific sources of information to make time-constrained decisions. Therefore, the twenty-first century is marked by a growing demand for bandwidth in wireless communications, as it allows users to communicate and access daily applications even from remote areas. Up to the present time, numerous breakthroughs in wireless communications were accomplished but mainly using the radio portion of the electromagnetic spectrum, which made RF to take the central role in today's communication systems. However, RF technology is a victim of its own success. Due to the tremendous increase in the number of mobile devices, RF technology cannot cope much longer with this market demand and will eventually reach a saturation point.

VLC is a recently appealing technique in the field of wireless communications that intends to complement RF technologies and is sought by many researchers as a viable alternative. VLC based on Light Emitting Diode (LED) takes advantage of these solid-state devices superior modulation capability to transmit data while assuring their lighting functionality. This work addresses the problem of achieving high bandwidth in a DLL design for OFDM based VLC broadcast systems and is inserted in a funded project called VLCLighting. The main objective of this dissertation work is to implement an efficient DLL in a Microblaze soft processor in a FPGA and to study its usage in a broadcast VLC system for lighting systems. Since two value added services were identified in the VLCLighting project, the proposed DLL aims at furnishing the adequate means to fragment and route those services requests while maintaining a continuous transmission flow that assures lighting and transceiver functionality. This work proposes a DLL design that was inspired in DVB and project OMEGA systems, able to describe the required amendments to fullfill VLCLighting goals.

Contents

List of Figures

List of Tables

Acronyms

API	Application Programming Interface
AWGN	Additive White Gaussian Noise
AXI	Advanced eXtensible Interface
BER	Bit Error Rate
BLER	Block Error Rate
CDMA	Central Direct Memory Access
CRC	Cyclic Redundancy Check
DAC	Digital-to-Analog Converter
DCO-OFDM	DC Biased Optical Orthogonal Frequency-Division Multiplexing
DLL	Data Link Layer
DMA	Direct Memory Access
DVB	Digital Video Broadcasting
DVB-S	Digital Video Broadcasting - Satellite
DVB-T	Digital Video Broadcasting - Terrestrial
DVB-RCT	Digital Video Broadcasting - Return Channel Terrestrial
EDK	Embedded Development Kit
FEC	Forward Error Correction
FER	Frame Error Rate
FPGA	Field-Programmable Gate Array
GSE	Generic Stream Encapsulation
HB-LED	High Brightness - Light Emitting Diode
HDL	High Description Languages
HDR	High Data Rate

HLS	High-Level Synthesis
IP	Intellectual Property
IRC	Infra-Red Communication
ISI	InterSymbol Interference
LED	Light Emitting Diode
LLC	Logical Link Control
LOS	Line-Of-Sight
MAC	Medium Access Control
MDR	Moderate Data Rate
MPEG-TS	MPEG Transport Stream
NLOS	Non-Line-Of-Sight
ODAC	Optical Digital-to-Analog Converter
OFDM	Orthogonal Frequency-Division Multiplexing
OSI	Open Systems Interconnection
OWC	Optical Wireless Communications
OWMAC	Optical Wireless Medium Access Control
OWLLC	Optical Wireless Logical Link Control
PLB	Programmable Logic Block
QPSK	Quadrature Phase-Shift Keying
RF	Radio Frequency
RS	ReedSolomon
RTL	Register-Transfer Level
SDK	Software Development Kit
SNR	Signal-to-Noise Ratio
VLC	Visible Light Communication
XPS	Xilinx Platform Studio

Chapter 1

Introduction

Wireless technology revolutionized our contemporary life providing the perfect means to globalization. With today's fast-paced society and its demands for higher data rates and mobile information technologies, wireless communications have been skyrocketing. We rely on these systems on a daily basis from our job to even our simplest everyday tasks. It influences all aspects of our life by bringing a new level of convenience with a continuous connection to our entertainment, educational and work applications. Therefore, the twenty-first century is marked by a growing demand for bandwidth in wireless communications, as it allows users to communicate and access those applications even from remote areas.

For over a century, the development of wireless technologies had an exponential increase. It all started when Nikola's Tesla invented the Tesla Coil, which he proposed could be used for transmission of information. Tesla's later demonstration, of a remote-control boat together with Marconi's first Atlantic wireless transmission, broke new ground in the communications field. Since then, numerous breakthroughs in wireless communications were accomplished but mainly using the radio portion of the electromagnetic spectrum. In the light of these achievements, Radio Frequency (RF) has taken the central role in today's communications systems. But as we see, a tremendous increase in the number of mobile devices and a growing demand for bandwidth, RF spectrum has become saturated, and new promising alternatives began to be sought. One of the most promising alternatives is Visible Light Communication (VLC), which is a rapidly developing field that benefits from an unregulated part of the electromagnetic spectrum, wide bandwidth and absence of electromagnetic interference.

Solid-State Lighting has advanced tremendously in the last fifty years and is beginning to unseat traditional illumination systems. This green lighting source has become a serious competitor of incandescent, fluorescent and sodium-vapor light sources, as Light Emitting Diode (LED) technology provides a reduce power consumption, higher life time and a more environment-friendly solution that has no lead and no mercury. As LED flourishes at a very fast pace, VLC takes advantage of its superior switching capability to transmit data, i.e., it takes advantage of LED fast switching characteristics to offer wireless communication. With the ramping up of LED technology and the invention of VLC systems based on them, we can use the lighting infrastructure already available and build the mentioned added value making them more efficient and economical than RF solutions.

VLC is a recently appealing technique in the field of wireless communications. In this technique, as the name suggests, data transmission is achieved through the use of the visible spectrum and it is a subset of Optical Wireless Communications (OWC). Artificially created

or naturally available, the optical radiation waves in this abundant electromagnetic spectrum have never been truly exploited for their value. It encompasses the frequency range from the 384,6THz up to 784,5THz that is the portion of the electromagnetic spectrum visible to the human eye.

The aim of this work is to propose a Network Layered Architecture for an Orthogonal Frequency-Division Multiplexing (OFDM) based VLC system. Throughout this document a DLL design and its implementation for a Microblaze in a FPGA will be in the spotlight of discussion. This work is part of a funded project called VLCLighting that is the result of a joint collaborative effort of different research projects with other master dissertations associated.

The current chapter presents the main motivations for researching a DLL for VLC, describing the importance of such technology while introducing the taken steps for the proposed Network Layered Architecture. Afterwards, the objectives will be explained and a brief description of the organization will be presented.

1.1 Overall Scenario

The present dissertation "DLL architecture for OFDM based VLC transceivers and its implementation in FPGAs" proposes and studies a DLL design in FPGA for Visible Light Communication broadcast application. This dissertation work takes place in the Integrated Circuits Group of the Telecommunications Institute of Aveiro and the proven expertise of this group in projects like VIDAS and LITES provided a set of conditions to support the conceptual framework and accomplish the project where this work work is included. This dissertation is inserted in the project VLCLighting (Visible Light Communications for LED based Public Lighting Systems) that was funded by FCT/MEC through national funds and co-funded by FEDER PT2020 partnership agreement under the project UID/EEA/50008/2013.

Project VLCLighting is a VLC project intended to provide broadcast services in public street lighting infrastructures. VLCLighting aspires to fulfill today's bandwidth need by redefining the existent public lighting with intelligent LED systems that can provide information access. VLCLighting is a joint collaborative project of different research efforts, and it is composed by an innovative Optical Digital-to-Analog Converter (ODAC) Front-End, DC Biased Optical Orthogonal Frequency-Division Multiplexing (DCO-OFDM) modulation, a Channel Encoder with Forward Error Correction (FEC) techniques and by this work proposal of a DLL. This project has identified two types of value added services to be introduced in public lighting systems. It identified a Moderate Data Rate service to provide adequate means for control and management, advertising and infotainment services and a High Data Rate service for video broadcast.

During the elaboration of this dissertation, projects like the OMEGA and the Digital Video Broadcasting (DVB) were part of the literature. Project OMEGA provided valuable insight into Medium Access Control (MAC) and Logical Link Control (LLC) protocols with their Optical Wireless Medium Access Control (OWMAC) and Optical Wireless Logical Link Control (OWLLC) specifications for Infra-Red Communication (IRC) and VLC systems. On the other hand, DVB, namely Digital Video Broadcasting - Return Channel Terrestrial (DVB-RCT), contributed with a broadcasting system behaviour overview, as well as with a baseline specification of return channel with their DVB-RCT MAC architecture proposal. These insights supported and inspired this proposed DLL structure design along

with its functionalities.

1.2 Dissertation Motivation

The realization of a DLL for a VLC system comprises several aspects. VLCLighting project has identified two types of value added services to be introduced in public lighting systems, a Moderate Data Rate (MDR) and a High Data Rate (HDR). The proposed DLL aims at providing the adequate means to fragment and route these data while maintaining a continuous transmission flow that assures the lighting and transceiver functionality. However, VLC broadcast poses several challenges to DLL design that need to be taken into account. The need for an optimized DLL raised as a means to comply with VLC limitations, as well as, project constraints that could not be entirely satisfied without a protocol that ensured data packaging and prioritizing from the multiple services.

After, It was found that not a lot of research has been done to develop a VLC outdoor broadcast system with a reliable DLL protocol. It was only found VLC research that focuses on indoor systems where environmental conditions can be conveniently controlled. From the review of the literature, it became evident that work should be done to look into the possibility of designing a modular architecture containing a DLL that could fit the present outdoor lighting infrastructures. Therefore, the objectives of the research presented in this dissertation led to a planned methodology mentioned in the following section.

It is important to remark that this work is only a preliminary step into a DLL standard for VLC based on OFDM systems that has not been established yet and presents many opportunities of improvement.

1.3 Methodology

The main objective of this work is to explore the feasibility for an efficient DLL implementation in FPGA and its usage in a broadcast VLC system for a public lighting system application.

In the first place, a study was carried out to identify possible challenges and understand the concepts of VLC systems, especially on outdoor scenarios. Secondly, it was made a study and research of MAC and LLC sublayers for VLC broadcast systems. It was found that in project OMEGA it was proposed an OWLLC and OWMAC for bidirectional VLC. On the other hand, DVB-RCT defined the MAC sublayer for digital terrestrial television. Based on this work literature, a DLL architecture that handles multiple broadcasting services was designed. The functions required to be programmed and their functionalities were characterized bearing in mind general project's needs. After this, study, further characterization was done to conciliate DLL with FEC techniques, efficiencies and to assure maximum performance for the project. In the next phase, DLL functions design with their respective programming structure and flowcharts were created.

This work proposes a DLL implementation in a FPGA, therefore Spartan605 Evaluation Kit was studied. The FPGA implementation started with the identification of its limitations. After this, embedded system tools and flows provided in the Xilinx Embedded Development Kit (EDK) were used for developing a system based on MicroBlaze embedded processors in Spartan605. Subsequent C++ programming of the DLL functions was accomplished with the Xilinx Software Development Kit (SDK). Finally, the performance of the proposed DLL was

analysed and several tests with different Direct Memory Access (DMA) configurations and different memories were examined.

In the process of investigation and realization of the proposed work, several software tools were used at the Instituto de Telecomunicações of Aveiro. These tools included Windows 7 Professional operating system along with Microsoft Office 2013 applications, and MatLab 2012a. The FPGA implementation was accomplished with Xilinx ISE Design Suite 14.5, namely with Xilinx Platform Studio (XPS) and Xilinx SDK.

1.4 Dissertation Structure

Now that the reader is introduced to the VLC overall scenario and this work motivations, further explanation of VLC systems and the proposed DLL will be discussed further on.

Chapter 2 summarises the technical background necessary for the project. VLC and basic concepts of LEDs are discussed briefly while giving to the reader their historical context. This chapter also presents and explores the proposed Communication Network Layered Architecture and some of the usual VLC applications.

Bearing in mind the unidirectional broadcast communication characteristic of VLCLighting project, the design of the DLL had numerous constraints that will be discussed in Chapter 3. This Chapter introduces the DLL concepts to provide the reader with full description of its functionalities behaviour.

Chapter 4 presents the embedded system implementation for the DLL. Basic concepts on FPGAs and the studied functionalities is presented, so full embedded process flow and DLL application designs could be understood. Finally, full analysis of DLL programmed functions concludes the discussion.

A set of test benches were performed to evaluate the programmed DLL behaviour, and these can be seen Chapter 5. Since this dissertation study major aims at video broadcast, different memory usage impact, as well as, different DMA configurations were the primal focus of this chapter. Chapter 5 also presents different time measurement techniques and DLL error handling capability.

Chapter 6 presents the final discussion and conclusions, as well as some future work guidelines.

1.5 Contributions

The proposed architecture along with an initial analysis of the DLL functionalities were used as part of a published article and talk at the ConfTele 2015 - 10th Conference on Telecommunications Aveiro, Portugal on September 17, 2015. The article VLCLighting - A Collaborative Research Project on Visible Light Communication "describes a collaborative research project on Visible Light Communications for lighting infrastructures. It is being developed by the Integrated Circuits and Mobile Network groups in Instituto de Telecomunicações, Aveiro site, and expects to deliver a VLC demonstrator transmitting video and data in real-time by the end of 2016. Another main goal is to develop this system to be modular in order to enable collaboration with other groups with interest in this field, offering the academic community a real-time test bed to evaluate the performance of different modules, algorithms and optical front-ends, which is currently not available" [1].

In this dissertation framework, a poster presentation in Students@deti was also performed to both students and teachers of the Department of Electronics and Telecommunications of the University of Aveiro.

Chapter 2

Technology context

Light is an indispensable part of our lives that we often forget. Light plays a central role in our well-being, as well as, all living beings. We rely too heavily on our eyes to gather all quotidian information and to perceive the world around us. Visible light has an enormous impact on our human experience, considering that we interpret the surrounding environment predominantly through vision. The interest in using this essential part of the electromagnetic spectrum as a communication medium started many years ago when ancient civilizations used fire beacons and smoke signals to send messages.

In modern society, electric lighting is taken for granted, because nowadays it is the most common form of artificial lighting but it was not always like this. Electric lighting revolutionized the world and with the later appearance and developments in LED technology, research in a new and promising VLC technology began. The ubiquitous LED with its recent ability to achieve meaningful data rates for today's communication needs made VLC based on LED a recently pervasive technology. This technology goal is to add extra value of data transmission to all present lighting systems and to promptly reply to the massive information demand.

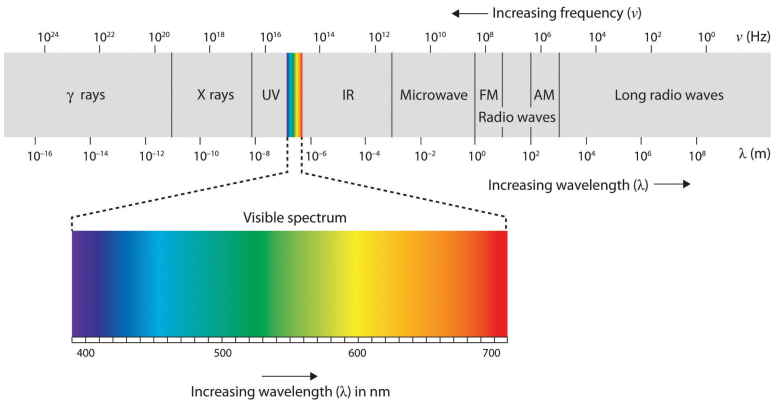

Figure 2.1: Electromagnetic Spectrum [2]

As mentioned before, VLC is a subset of OWC where data is transmitted using the visible light spectrum that is within 380nm and 780nm [3]. These wavelengths correspond to 384,6THz up to 784,5THz, as we can see in the Electromagnetic Spectrum figure 2.1.

Nowadays RF communications are the most popular technology, but the radio spectrum is saturated and cannot answer the demands of the consumers. Due to this, the visible

spectrum is the viable solution that provides a much wider spectrum (Radio spectrum = 300GHz while Visible Spectrum =400THz). Besides improved capacity, the visible spectrum provides more efficiency because radio waves consume a lot of energy while VLC is highly energy efficient. Another advantage is the availability because radio waves cannot be used in some sensitive scenarios like aircrafts due to interferences with other devices. While radio waves can penetrate walls, visible light cannot, presenting a useful resource to improve security in communications. The last advantage, but not least important, is human health, while the transmission of power radio waves cannot be increased over a certain level because there are serious health risks for humans, "VLC is considered to be harmless for the human body even while increasing the transmission power"[4]. Even though VLC has all of these advantages, it has some drawbacks and the most important one is the need to coexist with the existing lighting systems, while not interfering or being interfered by radio based wireless technologies.

2.1 VLC motivation

The worldwide spectrum crunch refers to the potential lack of RF spectrum required to support the social dependence upon wireless systems. This spectrum crunch represents a profound risk in the future of wireless networking and it persuaded many researchers to explore new alternatives.

Figure 2.2 illustrates the USA frequency allocations of the radio spectrum and it should give a better idea to the reader of the worldwide spectrum crunch.

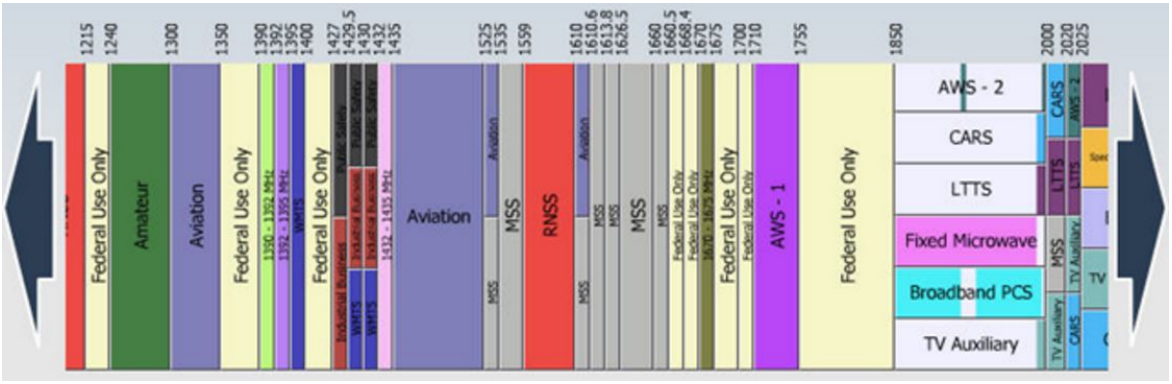

Figure 2.2: USA Frequency Allocations [5]

ANACOM is the national regulatory authority for communications in Portugal that plans and supervises the allocation of spectrum resources for many wireless applications, including radio and television broadcasting, land mobile service and satellite service. Each application is given a frequency band in which it is allowed to operate to ensure an efficient use of the available radio spectrum and to prevent interferences. From the 2015 ECO proposal([6]) and ANACOM's national table of frequency allocation [7], it is quite clear that the radio spectrum is very crowded. At the same time, the upward trend of wireless devices, evident in Figure 2.4, is one of the primary contributors for the global mobile data traffic growth (Figure 2.3). Cisco predictions estimate 24,3 Exabytes per month of mobile data traffic by 2019 and they confirm that wireless communications face a potential radio spectrum deficit. The stated predictions support the uptake of new wireless technology alternatives like VLC and they are the true

motivation for its research.

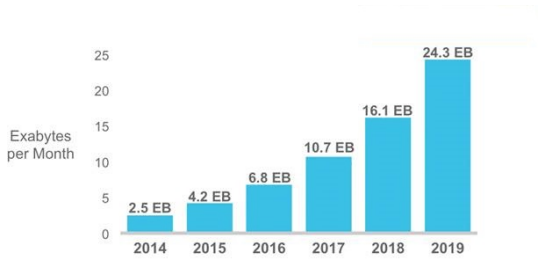

Figure 2.3: Global Mobile Data Traffic [8]

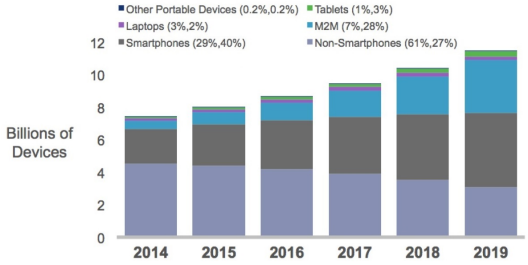

Figure 2.4: Global Mobile Devices [8]

In addition to the radio spectrum crunch, interference from the simultaneous use of a frequency band is also a concern for many existing wireless systems. The usage of mobile devices on airplanes, which can directly affect safety, is a prime example of radio interference risks. Until recently RF technology uses on airplanes was prohibited because wireless devices can cause interferences to the aircrafts navigation and communication systems. RF interference is also a large impediment to the success of wireless healthcare because it can have debilitating effects on medical telemetry systems that carry vital data and can also lead to unpredictable operation of medical devices.

To present the reader with the whole picture of VLC systems, VLC technology limitations will also be on the focus of this motivation section. Since VLC is a light-based technology, its major drawback is it inability to penetrate opaque objects. Although this confination of the information can be a privacy advantage in more secured scenarios, it can also be a liability in Line-Of-Sight (LOS) based systems. In order to use VLC technology for indoor, ambient noise produced by light emanating from both natural and artificial sources, needs to be taken into account. The interference produced by artificial ambient light sources like fluorescent and incandescent lamps can cause a significant degradation of these systems performance. The optical medium usage in outdoor scenarios also struggles to work flawlessly in adverse environmental conditions like fog and rain while trying to cope with external interferences introduced by sunlight that can lead to photodetector saturation problems. These restrictions, questions VLC usage for large-scale delivery of data transmissions and present significant challenges in this promising technology research.

Both indoor and outdoor VLC systems can have LOS, Non-Line-Of-Sight (NLOS) or hybrid architectures that present different behaviour characteristics. VLC nature favours the LOS architecture usage due to the existence of a direct communication channel between the emitter and receiver. NLOS, on the other hand, must rely on the reflections from the surrounding environment making them a more usual indoor solution. This electromagnetic propagation characteristic occurs when a full obstruction exists between the receiver and emitter. The hybrid solution tries to get the best from both worlds and chooses the best solution at that moment.

2.2 VLC History

The history of VLC dates back to ancient civilizations. In those civilizations fire beacons were fires lit on hills or high places used either as lighthouse for sea navigation through

dangerous coastal areas or for land signaling, when enemy troops were approaching. In Ancient China, soldiers had a set of communication signals that later were also used in the Great Wall. They would use fire beacons, flags and drum beats to warn other soldiers towers of enemy actions [9]. The first recorded use of the heliograph comes with Ancient Greeks in 405BC where they used polished shields to signal in battle [10], as seen as in Figure 2.5.

Figure 2.5: Light Communication using Combat Shield [11]

The hydraulic telegraph (Figure 2.6) was invented by a Greek named Aeneas in 350BC, which was a fast communication system with fairly detailed information over long distances. At that time, fire was used to indicate danger or that an objective has been accomplished but without any level of detail. This detail limitation was overcome by having a system of water-filled vessels containing rods that contained agreed-upon messages. Both groups of the communication needed to be in line-of-sight and upon agreement, with torch lighting, they would pull the plug from the bottom of the water-contained vessel. As the water drained, different messages on the rod would be revealed and again with torch lighting signal the receiver could know which message to read on the rod [12].

Figure 2.6: Hydraulic Telegraph of Aeneas [12]

Figure 2.7: Heliograph [13]

The Lighthouse of Alexandria was the first tower used as a lighthouse. The lighthouse was constructed in the 3rd century BC and it was one of the tallest man-made buildings for many centuries. It was responsible for navigational guidance of maritime pilots. In the

2nd century BC Polybius wrote a communication and data encryption system, also known as the Polybius square. The system divided the letters of the alphabet in five parts written in a tablet with 5 letters each, but due to modern alphabet having 26 letters the I and J were combined. Upon the raise of two torches, the transmitter waits for the reply from the receiver to confirm the communication. After that, the transmitter raises the first set of torches signaling which tablet of the five to consult. Next, he raises the second set of torches that indicates which letter to write down and following the same principle the transmitter sends the following letters. Even being accurate this process was proven to be tedious [11]. For many years visible light communication was made essentially with fire, an example of this is the Byzantine Empire that in the 9th century used a system of fire beacons to transmit messages from the border to their capital (Constantinople). In 1821 the German scientist Karl Gauss found a mean to mirror a controlled beam of sunlight to a distant station even though this device (heliotrope) was meant to be used in geodetic surveys [14]. This device was the predecessor of the first widely accepted heliograph made by Sir Henry Christopher Mance in 1869 that added a movable mirror that later could be used to signal Morse code. The heliograph, seen in figure 2.7, provided a very mobile element in the British Signal Service operated where two mirrors were used to gather sunlight and send it to a prearranged spot as a coded series of short and long flashes [15].

Later, in 1880 Alexander Graham Bell invents the photophone (Figure 2.8) which is a device that transmits sound on a beam of light. On June 3, Bell conducted the world first wireless transmission from the top of the Franklin School in Washington, D.C. “Bell’s photophone worked by projecting the voice through an instrument toward a mirror and he observed that vibrations in the voice caused similar vibrations in the mirror”. Bell directed sunlight into the mirror, which captured and projected the mirror’s vibrations. These vibrations were transformed back into sound at the receiving end of the projection. The photophone functioned similarly to the telephone, except that the photophone used light as a mean of projecting the information and the telephone relied on electricity [16].

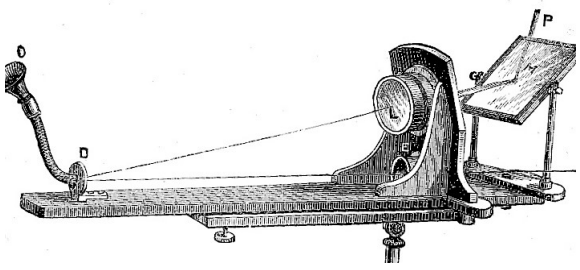

Figure 2.8: Photophone [17]

An American Engineer named Eugene Polley in 1955 invented the Flash Matic, which was the first-ever wireless TV remote control. Polleys Flash-Matic consisted in pointing a visible beam of light at photo cells in each corner of the television screen. The viewer used a highly directional flashlight to turn the picture and sound on or off, or to change the channel in a clockwise or counter-clockwise direction [18]. The discovery of optical sources like LASER (Light Amplification by Stimulated Emission of Radiation) in the 1960s was an important step forward in the history of VLC. This breakthrough was the base for future projects such as the NASAs Lunar Laser Communication Demonstration (LLCD) that is a project for deep space communication using visible light. In 1976 we see the introduction of LED in VLC

when Thomas P. Pearsall research revolutionized telecommunications. Since the 1960s LEDs have doubled their light output and power efficiency every 36 months. This behavior is known as Haitzs Law shown in Figure 2.9. Haitzs Law said that every decade, the cost per lumen (unit of useful light emitted) falls by a factor of 10 and the amount of light generated per LED package increases by a factor of 20, for a given wavelength (color) of light [19].

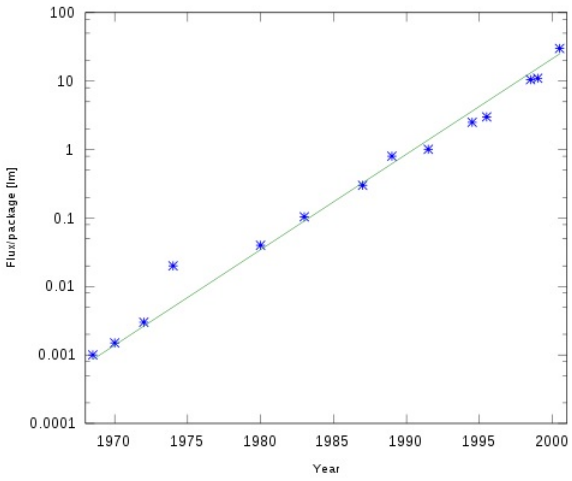

Figure 2.9: Haitz Law [19]

Since 2003 there have been various research efforts in VLC for data transmission using LEDs and the most important ones were made by the Nakagawa Laboratory, Smart Lighting Engineering Centre, Omega Project, D-Light Project, UC-Light Centre and work at Oxford University [4]. Due to VLC importance a Visible Light Communication Consortium was formed in Japan in 2004 in order to realize safe, ubiquitous telecommunication system using visible light through the activities of market research, promotion, and standardization. This VLCC later proposed two visible light standards to JEITA (Japan Electronics and Information Technology Industries Association) and the two proposals became JEITA standards in June 2007 [20]. With VLC technology being researched all over the world, IEEE saw the need to standardize it. So, in January 2009 the IEEE 802.15.7 VLC Task Group completed a PHY and MAC standard for Visible Light Communications [21]. More recently, we see Organic Light Emitting Diodes being used to achieve 10Mb/sec and to still have a great focus in VLC research nowadays [22]. Then, in 2010, Siemens researchers with the help of Fraunhofer Heinrich Hertz Institute improved the record of wireless data transfer to 500 Mbit/s using White LED light [23]. In July 2011 Harald Haas made a live public demonstration of VLC technology at TEDGlobal 2011 by transmitting high-definition video with a standard LED lamp [24]. This presentation made VLC technology even more worldwide known. Recently, in 2015, Fraunhofer HHI developed a VLC demonstrator that exploits a much higher bandwidth of up to 180 MHz and achieves a transmission rate in laboratory experiments of over 1 Gbit/s per single light frequency. This means that their newly prototype with white RGB LEDs can get speeds of up to 3Gbit/s.

2.3 Types of VLC

In order to illustrate the reader with general VLC applications, this section will be divided in three subsections that will discuss indoor VLC, outdoor VLC and underwater VLC scenarios.

2.3.1 Indoor VLC

White LEDs are commonly used as replacements for incandescent lamps and as their performance improves, these devices will become candidates for replacement of general illumination. These solid-state sources can be modulated at superior rates where incandescent or fluorescent alternatives are very hard to be modulated. Thus offering the possibility of broadcasting information at the same time as providing illumination. The indoor practical applications of VLC indoor are numerous. For instance, it can be used to broadcast TV signal and other multimedia contents, as well as, provide internet access. Other possible application is indoor navigation system by preinstalling LED ceiling lightning that has an ID encoded. Other indoor VLC applications are mines and aviation where RF communications are undesirable.

2.3.2 Outdoor VLC

In outdoor VLC there are many external light noise sources like sun light, road/street lights that need to be taken into consideration. In long distances this kind of environment poses a major difficulty to be dealt, because fog, smoke and temperature variances are major problems. Since most of outdoor VLC systems are short range communications, this atmospheric restrictions do not impact so much. Another limitation is that most outdoor VLC systems need line-of-sight access which in some urban areas with obstacles or some natural barriers can provide issues. Outdoor VLC applications are already being developed for traffic signs, communication between vehicles, traffic updates, in-car entertainment and outdoor advertising.

2.3.3 Underwater VLC

Due to underwater RF communications limitation, VLC is a successful choice amongst researchers as an alternative for underwater VLC. "The underwater environment provides a promising area of application for wireless optical communications; here visible light experiences significantly lower attenuation than the remaining electromagnetic spectrum" [25], especially in the green wavelength. A system capable of interactive communications between divers that sends the voice from a diver and sends to another diver using light is already being developed by a joint group of 21 Zamami Co, RISE Co, Keio University and Nakagwa Laboratories. This promising VLC project together with VLC potential for underwater communication, makes the use of this technology ideal for underwater exploration.

2.4 VLC Architecture

A basic VLC system consists of a light source, free space as the propagation medium and a light detector. Information, in the form of digital or analog signals, is input to electronic

circuitry that modulates the light source. The source output passes through an optical system (to control the emitted radiation, for instance, to ensure that the transmitter complies with eye safety regulation) into the free space. The received signal then comes through an optical system that has an optical filter able to reject optical noise and a lens system or concentrator that focuses light on the detector. The resulting photocurrent at the end of the photodetector is then amplified before the signal processing electronics [26].

Figure 2.10: VLC Concept

2.4.1 VLC Emitter

Visible Light Communication Emitters typically consist in a visible solid-state emitter that can be either an LED or a semiconductor LASER, depending on the application. LED stands for Light-Emitting Diode and it is a semiconductor p-n junction device that gives off spontaneous optical radiation, which in another words, is a diode that emits light when electricity passes through. LASER stands for Light Amplification by Stimulated Emission of Light meaning that a source light is amplified within a gain medium and focused in a target area. While LED, like most sources of light presents an incoherent light emission, LASER presents us a coherently light emission source and as a result there is minimal loss of energy along the beam and the target area. Coherent light features all the individual light waves precisely lined up with each other and working in perfect unison. Usually when the emitter acts as a communication transmitter and an illumination device at the same time white light LEDs are used. In the Figure 2.11 of the ACO-OFDM VLC transmitter, a data message (Digital Data) passes the channel encoder where a convolutional coding or Reed-Solomon algorithm is employed (further discussed in sections 2.7.2 and 2.7.2) and then, the signal is scrambled with the interleaving block (section 2.7.2). While channel coder main objective is to compute the algorithm so receiver then uses it in the channel decoder. It also aims at code correction in general purposes, where interleaving block aims code correction of burst errors. The interleaving block makes a more uniform distribution of the errors, so a word can be recovered even in the case of a burst error occurrence (a group of adjacent bits that are all affected). After scramble operation, modulation is the next step. The Quadrature Phase

Shift Keying (QPSK) adds two more phases (90° and 270°) allowing the transmission of two symbols per bit. Modulated waves with Quadrature Amplitude Modulation (QAM) are the combination of both phase-shift keying (PSK) and amplitude-shift keying (ASK) allowing an increase of bit-rate. For example, 16 QAM can represent four bits instead of just the two bits per symbol with QPSK. Further modulation is required and solutions like DCO-OFDM and ACO-OFDM were considered, being the last one seen in Figure 2.11 while DCO-OFDM can be seen in Figure 2.12. Both modulations are studied in greater detail in the section 2.7.3. OFDM signals are generally complex and bipolar, so the following blocks are responsible for making this signal real and positive. Considering the ACO-OFDM modulation, the first step is the use of Hermitian symmetry on the complex signal that feeds into the IFFT block [27]. This step ensures that the output of the IFFT is real. In the case of DCO-OFDM an additional step for adding a DC to obtain a unipolar signal would be needed, but in ACO-OFDM this is not needed. After having a real signal, the next step is the cyclic prefix block (CP) that ensures no intersymbol interference, because it adds a safeguard interval between the symbols to transmit. As a final step, the emitter clips the signal at zero level that removes the entire negative signal because in VLC it is physically impossible to transmit negative signals. The pilot block in the architecture is for measurements of the channel conditions and calculation of the equalization gain in the receiver. The physical layer of a VLC emitter diagram architecture with ACO-OFDM can be seen in Figure 2.11.

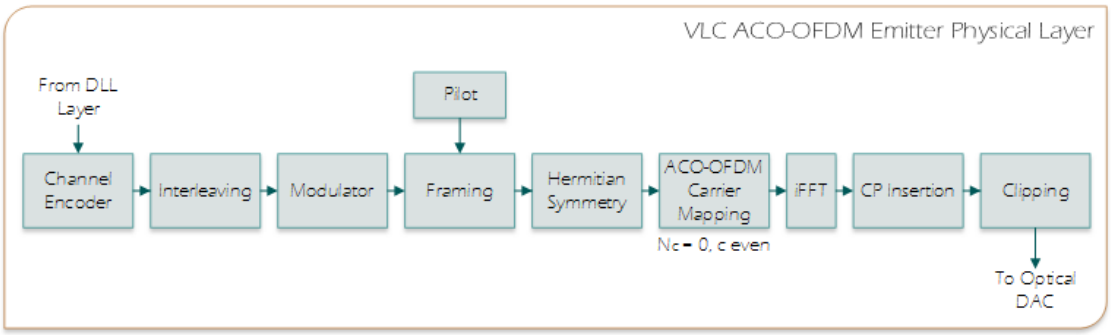

Figure 2.11: VLC PHY ACO-ODFM emitter architecture

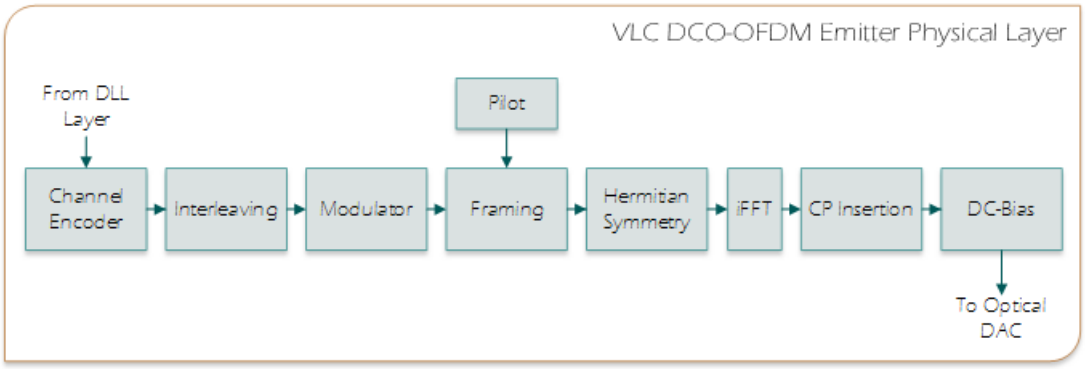

Figure 2.12: VLC PHY DCO-ODFM emitter architecture

2.4.2 VLC Transmission Channel

In an indoor environment the primary use of the VLC sources is to provide illumination. Therefore, the illumination and communication must be considered simultaneously. In a typical broadcast VLC system the receiver is below the ceiling where the light from the arrays propagates to provide a communication channel. There is a line of sight (LOS) path from each LED to the receiver, and also a diffuse path via the reflections from surfaces within the room. The LOS path strength can be calculated using a knowledge of the emission patterns of each source and a simple propagation model. There are a number of different ray-tracing approaches that can be used to simulate the diffuse path, but these are complex and time-consuming which makes the alternative approach that models the room as an integrating sphere a viable solution [28]. There are ambient light interference that affect the VLC channel such as incandescent lights, fluorescent lamps or the sunlight. "Incandescent lights emit high levels of infrared radiation, fluorescent lamps emit high levels in the visible spectrum range and the sunlight emit ultraviolet radiation and a considerable amount of infrared radiation" [4]. All these noise sources need to be addressed accordingly and they will be addressed in the next Chapter.

2.4.3 VLC Receiver

Visible Light Communication receivers typically consist of a collection optics and an optical filter, which collect light of the correct wavelength range. The resulting photocurrent is amplified, normally using a transimpedance preamplifier. The optical filter is used to reduce the effect of ambient light by creating a narrowband optical system that only allows the modulated source wavelength to be detected. In the case of VLC the modulated energy covers the entire visible spectrum, so that filtering will reduce both the ambient light noise from daylight and the desired signal [28]. One of the main parts of the VLC Receiver diagram is the photo-detector which normally is a PIN photodiode or an Avalanche photodiode that converts the optical signal into an electrical current for further amplification. Another important stage to be considered is filtering. Considering that many noise sources do not only affect visible light spectrum and due to PIN photodiodes spectral sensitivity that includes a large part of near Infrared Spectrum, an IR cut-off filter is used to filter the signal. Since all ambient light indoors and outdoors flicker, and this behavior is considered as noise to VLC it needs to be filtered. The interference from ambient lights is of additive nature to the modulated information and all the received optical signals are superimposed together [29]. The Front-End Receiver C12702-11 already has a band pass filter that rejects frequencies outside the range of 4 KHz up to 100MHz. After this, another band pass filter is required to remove the 50Hz noise originated from the other illumination appliances. Afterwards, a low pass filter is used to avoid the aliasing effect and an Analog to Digital converter is used to digitalize the signal. Cyclic Prefix remover is responsible for removing the safeguard interval between the received symbols for further conversion, by the FFT, to frequency-domain. The ACO-OFDM Frame Demapping removes the sub-carriers that are set to zero (odd) and passes the rest to the Hermitian symmetry rejection that ensures that the output does not have the second half of sub-carriers which contain the conjugates in inverse order. Deframing is responsible for selecting the sub-carriers that contain data from the other ones that contain pilot or synchronization purposes. After this, the Channel Estimation block uses the occasionally sent Pilot Block information to analyze the channel conditions and equalize it with the Equalizer

blocks, if necessary. Then the demodulator block is responsible for demodulation the signal. After demodulation the scrambling is removed by the Deinterleaving block and the Channel Decoder uses the result from the algorithm, done by the emitter, of the Channel Encoder to correct bit errors that might have occurred. After all these PHY layer steps and after the MAC Management layer at the output we have a digital data output that should be equal to data input in the VLC emitter. A full diagram architecture of the Physical Layer of ACO-OFDM VLC receiver can be seen in Figure 2.13. The DCO-OFDM receiver can be seen in Figure 2.14 which has a similar architecture regarding the presented ACO-OFDM with the exception of Carrier Demapping that is not required.

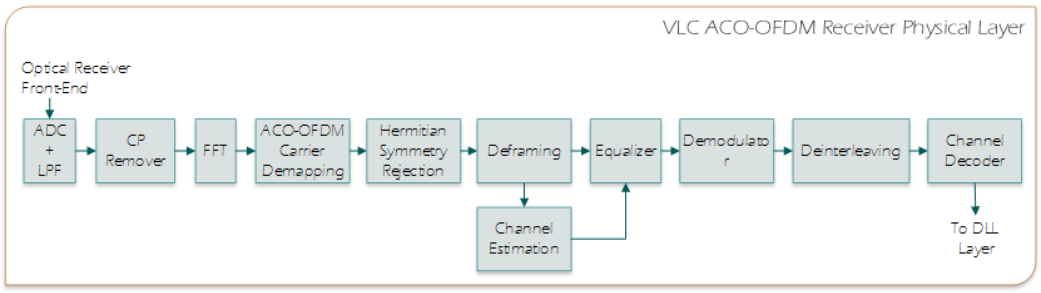

Figure 2.13: VLC PHY ACO-ODFM receiver architecture

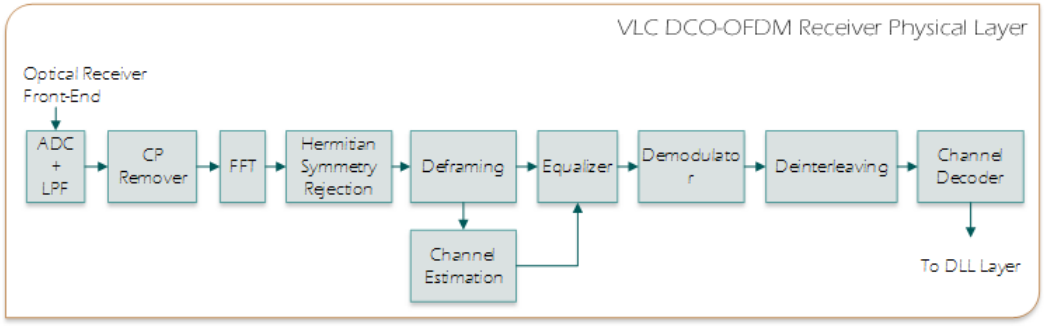

Figure 2.14: VLC PHY DCO-ODFM receiver architecture

2.5 Light-Emitting Diode

LEDs have come a long way in the last decades and the continuous advances in technology provided a great increase of LEDs luminous output. LED technology research and developments together with its very energy-efficient and long lifetime characteristics made it a viable solution for the lightning market . Due to thermal inertia both incandescent and fluorescent light bulbs are not able to carry information. LEDs are solid state devices which allows them to switch between on and off states at moderate frequencies and this characteristic can be explored to modulate information using visible light waves [30]. Typically there are two methods to create white light LEDs. Red, Green and Blue devices are combined in a single package and the white light is generated by mixing them in appropriate proportion. The advantage of this approach is to be able to tune the colour by altering the current to each device. However, the additional complexity of packaging and drive electronics has made this

approach to providing general illumination less attractive. The widely used alternative is to combine a blue LED with a coating of a phosphor that emits yellow light. The blue emission from the LED combines with the yellow from the phosphor to create white light which has the advantage of requiring only a single electrically driven source [28]. The comparison of both methods is shown in Figure 2.1. RGB LEDs have a bigger bandwidth and consequently achieve higher data rate but they require more difficult modulation techniques.

	RGB LEDs	Phosphor-based LEDs
Data rates	$\cong 100 Mbits/sec$	$\cong 40 Mbits/sec$
Price	More expensive	Low cost
Bandwidth	Higher bandwidth	Phosphor limits the bandwidth
Modulation	Complex	Low complexity

Table 2.1: RGB LEDs vs. Phosphor-based LEDs [4]

2.5.1 LED History

In 1907 Henry Joseph Round, personal assistant of Guglielmo Marconi, noticed the natural phenomenon called electroluminescence in a piece of Silicon Carbide [31]. Electroluminescence is the production of visible light by a substance exposed to an electric field without thermal energy generation [32]. Twenty years later, the Russian Oleg Losev observes and describes in greater detail light emission from zinc oxide and silicon carbide crystal rectifier diodes used in radio receivers when a current was passed through them. In the 1950s British experiments into electroluminescence using Gallium Arsenide led to the first modern LED, short word for Light Emitting Diode. In this decade an American physicist and inventor named William Shockley, co-inventor of the transistor in Bell Labs, files a patent for an infrared LED in 1951, Kurt Lehovec then patent a SiC visible light LED in 1952 and Rubin Braunstein and Egon Loebner patent a green LED in 1958. Even with all of these evolutions and all issued patents it is Nick Holonyak that is considered by most to be the pioneer of Light Emitting Diode technology. This American Engineer is responsible for creating the first practical visible-spectrum LED in 1962, while working at General Electric Company. This first LED was red and was made of GaAsP (Gallium Arsenide Phosphide) on a GaAs substrate [31]. This year marks the birth of the industrially-produced LED, costing 110,13€ each. After this date companies such as IBM uses them in circuit boards (1964) and Hewlett Packard integrates them in hand-held calculators (1968) [33]. As we can see in Figure 2.15 these LEDs were merely for indication since the light output was not enough to illuminate.

Figure 2.15: LED display in TI-30

With LEDs having the focus of many research Labs, M. Georges Craford invents the first

yellow LED and develops a brighter red LED in 1972. Four years later Thomas P. Pearsall creates the first high brightness and high efficiency LED for optical fiber telecommunications improving communications technology worldwide. Another breakthrough in LED technology is made in 1993 when Shuji Nakamura creates a high brightness blue LED using GaN (Gallium Nitride). This discovery quickly led him to design a white LED that went to the market two years later [34]. The development of LED technology after this was focused in improving its brightness making their efficiency and light output to raise exponentially. With this focus in mind, in 2006 the first LED with 100 lumens per watt was produced [34] making LED lightning a viable solution. Since this major breakthrough we can see examples like Torraca (Italian village) that was the first town to convert its entire illumination system to LED technology. This lead to a reduction in energy and maintenance costs by 70% (2007). Audi was the first car company to implement a full LED headlights in 2008 [35]. There are already efficient LED architectures to approach the target of 250lm/W for 3000K/80CRI and 225lm/W for 3000K/90CRI [36].

2.6 Photodiode

A photodiode is a semiconductor that converts light into electrical current making it suitable for accurate measurement of light intensity. When photons are absorbed by a photodiode a current is generated. Every photodiode has a dark current which is the current through the photodiode in the absence of light. This dark current is important to bear in mind because it represents the minimum detectable signal and a signal needs to produce more current then the dark current to be detected. Dark current is correlated with the operating temperature, bias current and the type of photodiode [27]. There are several principles of operation of photodiodes, being the PIN and APD the most known ones.

2.6.1 PIN

PIN photodiode is the most widely used semiconductor detector and it is a diode with a wide undoped intrinsic semiconductor between a p-type semiconductor and a n-type semiconductor. The junction is reversed biased by an external source and the bias voltage appears almost entirely across the intrinsic section. This is because the intrinsic section is free off charges and therefore is much more resistive than the p and n sections, which forms a strong electric field. When photons hit this section they make valence band electrons to jump into the conduction band leaving a hole behind, which promotes the creation of photo-generated carriers. The created carriers in the middle section drift due to the strong electric field, making electrons to move towards the n-region and holes towards the p-region. This phenomenon causes a photo-current to flow into the circuit [37]. PIN photodiode is capable of operating at a very high bit rate exceeding 100Gbps. However the commercial available ones only offer bandwidths up to 20GHz and this is mainly due to package limitations [27]. Although, the previous presented results are usual for optical fiber communications, VLC systems give more importance to receiver sensitivity. Therefore, APD photodetectors that provide better sensitivity will be discussed in the following section.

2.6.2 APD

APD can be thought of a photodetector that provide a built-in first stage of gain through avalanche multiplication. By applying a high reverse bias voltage, APD shows an internal current gain effect, normally around 100, due to impact ionization called avalanche effect. However, some silicon APD employ alternative doping that allows an even greater voltage to be applied (more than 1500 V) before breakdown is reached and hence a greater operating gain, which can be more than 1000. In general, in APD the higher the reverse voltage is, the higher the gain. Avalanche photodiode is a highly sensitive photodiode that operates at high speeds and high gain by applying a reverse bias. These optical detectors are the preferred ones for high bandwidth applications where the wavelength lies between 400nm and 1100 nm and where high speed and low optical power are required. APD photodiode has worst Signal to Noise ratio when compared to PIN photodiodes and presents an increased complexity of operation. However the main difference with the PIN diode is that the absorption of a photon of incoming light may set off an electron-hole pair avalanche breakdown. This feature gives the APD a much greater sensitivity when compared with PIN [38].

2.7 Communication Network Layered Architecture

The VLCLighting project arquitecture that will be shown in Section 3 is composed by a DLL, a Channel Coding and modulation. Therefore, this section will present the reader with a brief overview on their propose and their usual usage in VLC systems.

2.7.1 Data Link Layer

DLL layer goal is to provide services to the higher layer that enable transparent and independent control of the transmission and reception of data from the physical layer. The detailed operations within the DLL layer are hidden from the above layers. Such operations can be frame delimiting and recognition, addressing of destination stations, conveyance of source-station addressing information, collision resolution (on shared broadcast links), protection against errors (generating and checking frame check sequences) and control of access to the physical transmission medium. The MAC sublayer within DLL is responsible for channel access control mechanism. Electrical multiplexing and optical multiplexing are two possible strategies to realize this multiple access control but there is also packet mode mechanisms of multiplexing. Electrical multiplexing techniques includes Frequency-division multiple access, Time-division multiple access, Space-division multiple access and Code-division multiple access. Optical multiplexing techniques include wavelength-division multiple access (WDMA) and space-division multiple access (SDMA). Packet mode mechanisms include Carrier sense multiple access with collision avoidance (CSMA-CA), ALOHA and Multiple Access with Collision Avoidance (MACA). Since this work will be focused on a broadcast system, as will be discussed further, where channel sensing is not possible, these will not be implemented.

The functions of the DLL are divided into Provisioning Management, Connection Management and Link Management. Provisioning management is responsible for time or sub-carriers selection. Connection management manages addresses from upper layer converting them into physical addresses. Link management is responsible for establishing the relation between the physical addresses and the location of its physical destination. On the other hand, MAC data service provides data transport services between the peer MACs. One important function of

DLL in VLC systems is the idle pattern that can be transmitted during medium access layer idle (no data transmission) for infrastructure light sources. This DLL idle state helps maintain visibility and flicker-free operation. The data and the idle pattern should have the same duty cycle to minimize flicker and these visibility patterns are not encoded in the PHY layer and do not have a frame check sequence (FCS) associated with them [39]. For instance, in the DVB-RCT, individual packets are organized in groups and constitute a mega-frame and each mega-frame contains exactly one Megaframe Initialization Packet (MIP). MIP is made of 4 byte header and a 184 byte data field and contain many important parameters, such as, the transport_packet_header, the synchronization_id, periodic_flag, sectionlength, pointer and maximum delay [40]. In this MIP there are also functions like enable_function, bandwidth_-function and tx_power_function that are used for their bidirectional case.

2.7.2 Channel Encoder

Convolutional Coding

Convolutional code is a type of error-correcting code that generates parity symbols via the sliding application of a Boolean polynomial function to a data stream. Unlike block codes, they involve the transmission of parity bits that are computed from message bits and the sender does not send the message bits followed by the parity bits, it only sends the parity bits instead.

Reed-Solomon

The Reed-Solomon algorithm is a type of block code, which means that encodes data in blocks. This algorithm is widely used in telecommunications and in data storage. It is also part of all CD, DVD readers and most barcodes, where it provides error correction and data recovery. It also protects telemetry data sent by deep-space probes such as Voyagers I and II. And it is employed by ADSL and Digital TV hardware to ensure data fidelity during transmission and reception. Reed-Solomon codes are BCH algorithms that use finite fields to process message data. They use polynomial structures to detect errors in the encoded data and they add check symbols to the data block, from which they can determine the presence of errors and compute the correct values. BCH algorithms offer precise, customizable control over the number of check symbols and provide simplified code structures, which makes them attractive for hardware implementations [41].

Concatenated coding

Concatenated coding is an error-correcting algorithm that is constructed from two simpler coding algorithms in order to achieve better performance at the cost of increasing complexity. Concatenated codes combine an inner code and an outer code and it was proposed by Dave Forney [42] as a solution to the problem of finding a code that with increased block length would exponentially decrease the error probability.

Interleaving

Interleaving is a technique to make forward error correction more robust, breaking up burst errors. This technique is ideal for burst errors correction that occur, for example, in

fast fading channels because it shuffles source symbols across several code words creating a more uniform distribution of the errors. Otherwise, if the number of errors within a code word would exceed the error-correcting code capability, it would fail to recover the original code word.

Puncturing

A punctured code is obtained by periodically deleting encoded symbols from ordinary encoded sequences, in order to increase the code rate. Puncturing is used to create the needed variable coding rates to provide various error protection levels to the users of the system. For example, in a Viterbi decoder that only operates on the metric of one input bit per encoded symbol instead of the higher numbers needed, and where the traceback length is usually five times greater than the constraint length, the puncturing ensures that the decoder has enough time-steps to properly decode the noisy input bits [43].

Pilot Carrier

Pilot carrier is a pilot signal transmitted over the communication system for supervisory, control, equalization, synchronization or reference purposes. In the proposed architecture pilot carrier is used for measurements of the channel conditions and calculation of the equalization gain and the phase-shift for each sub-carrier. Due to the presence of a DC component output in the receiver, which can affect the demodulation of digital signal, an orthogonal carrier pilot should be used. This simple method significantly reduces the bit error rate (BER) of data symbol detection [40].

2.7.3 Modulation Techniques

VLC technology uses simple intensity modulation and direct detection schemes, which requires the modulation signal to be real-valued and positive. Well-known modulation schemes from radio frequency communication can be adapted into the visible-light domain. Such examples include on-off keying (OOK), pulse-position modulation (PPM) and pulse-amplitude modulation (PAM) [44].

OOK

OOK transmits the higher state (bit 1) by turning on the light and transmits the lower state (bit 0) by turning off the light. When transmitting the bit 0, the light does not necessarily require to be turned off completely, but dimmed at a certain lower level relatively to the turning on state. OOK is the simplest Amplitude-Shift Keying modulation and it represents the digital data as the presence or absence of a carrier wave, which in the case of VLC is the presence or absence of light.

OFDM

OFDM is a method of encoding digital data on multiple carrier frequencies. OFDM allows equalization to be performed efficiently in the frequency domain and makes sure that available communication resources are always fully-utilized, because it adaptively encodes into different frequency bands. Conventional OFDM generates complex-valued bipolar signals,

which need to be adapted to become suitable for VLC. In order to have the time-domain waveform, real Hermitian symmetry is used. This symmetry allows all negative samples to be removed, and as a result a purely positive signal is obtained without any distortion to the useful information. This operation effectively maps the information symbols onto sub-carriers in different frequency bands. Due to complex-value nature, a real OFDM signal can be obtained, but reduces the system bandwidth by a half. However, the resulting waveform still has a bipolar nature (it has a positive and negative part). A valid approach to make a bipolar signal strictly non-negative is to introduce a direct current (DC) bias around which the original bipolar signal can vary, this is known as DC-Clipped optical OFDM(DCO-OFDM) [44]. Since OFDM signals have a high peak-to-average amplitude ratio, the required DC bias can be high. The power dissipation increases due to this bias and makes asymmetrically-clipped optical OFDM (ACO-OFDM), that requires no biasing, a strong alternative. ACO-OFDM clips the entire negative excursion of the electrical waveform to minimize average optical power. In ACO-OFDM the odd-indexed sub-carriers in the OFDM frame are modulated with information while even sub-carriers are set to zero [45]. More recently, a technique known as unipolar OFDM (U-OFDM) has been proposed and it takes a real bipolar OFDM signal and generates a unipolar signal by splitting every OFDM frame into two separate frames in the time domain. The first frame contains the positive time-domain samples and zeros in place of the negative samples and the second frame contains only the absolute values of the negative samples and zeros in place of the positive samples. Then at the receiver, the original bipolar frame can be obtained simply by subtracting the frame containing the negative samples from the frame containing the positive samples. The result of all these manipulations is a purely unipolar signal which requires almost no biasing and this leads to very significant electrical and optical power savings in comparison to DCO-OFDM [44]. Only a quarter of all subcarriers in an ACO-OFDM are utilized to carry information while U-OFDM frame sacrifices half of the information capacity in comparison to conventional OFDM. In ACO-OFDM, the capacity reduction comes from not utilizing the evenly-indexed sub-carriers, while in U-OFDM this comes from the splitting of each bipolar OFDM frame into two unipolar frames.

Chapter 3

Designing a DLL for VLC

The main objective of this dissertation is to study and implement a DLL for a broadcast VLC system. This work is part of a wider project called VLCLighting. Before going into the details of its implementation is necessary to provide the reader with an overview of the project characteristics and goals. This chapter will present VLCLighting architecture, the DLL implementation objectives, as well as, the DLL concepts, project constraints encountered, DLL functions description and their respective flowcharts.

VLCLighting architecture is composed by a modular system with a DLL, FEC, modulation and optical front-ends, as it can be seen in figure 3.1.

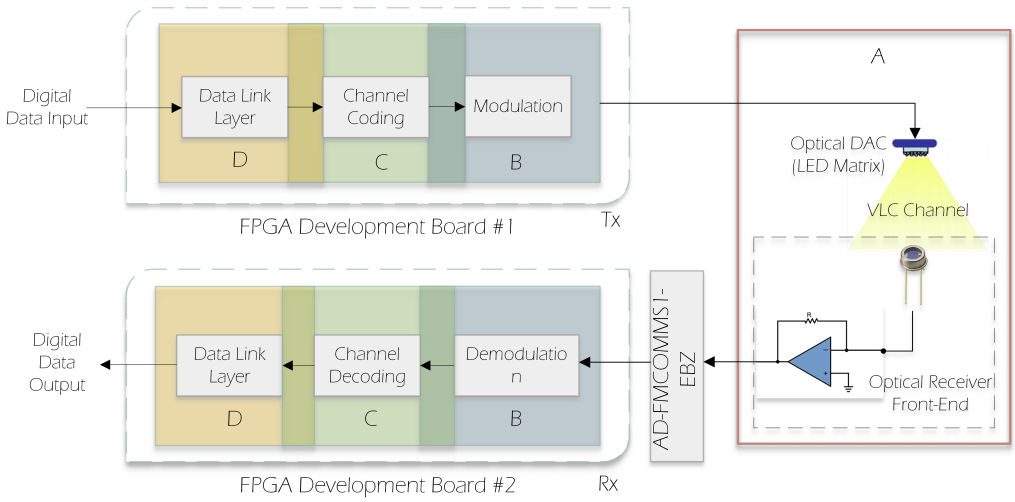

Figure 3.1: VLCLighting architecture

The optical emitter is an ODAC with 255 High Brightness - Light Emitting Diode (HB-LED) that ensures communication with an Hamamatsu optical receiver and two FPGAs were used to develop all the remaining functional blocks. The emitter front-end is an ODAC that offers 8 bits of resolution. To achieve the lighting functionality 255 HB-LEDs were used, where different digital bit weights were given to different number of LEDs. These 255 LED array is called an ODAC, which is a digital to analogue converter in the optical domain. With ODAC usage we can achieve further distances without using power LEDs and also mitigate the non linear effects on this solid state devices. The non-usage of power LEDs also benefits with

a larger bandwidth. LEDs are usually designed to be operated around a constant current, which makes its non-linear characteristics an issue in VLC systems where a DAC is used to send analogue information. With ODAC it is possible relax design constraints correlated to current driving the LEDs and its variation effect in luminous power. This allows to start thinking about the precision between one digital bit to its adjacent, which have an error of half of the least significant bit value. The receiver front-end is from Hamamatsu (C12702-11) and provides a $4KHz$ to $100MHz$ bandwidth with low noise and compact design.

A key part of the VLCLighting system that offers integration between all mentioned blocks is its asynchronous architecture. Using this architecture all blocks are independent, making their design and testing simpler. The act of adding or removing a single block is simplified due to the interface and synchronization abstraction. With this asynchronous implementation, reusage of hardware is also a possibility as well as high configurability parameters. However, it presents a higher complexity of each block implementation due to the need of additional control blocks, higher memory requirement and its initial delay.

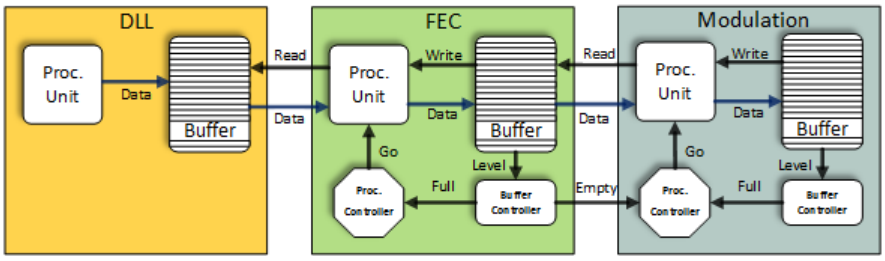

Figure 3.2: Tx Asynchronous architecture

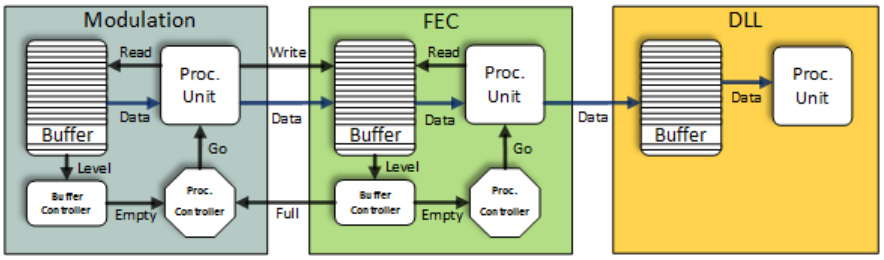

Figure 3.3: Rx Asynchronous architecture

The asynchronous architecture of the emitter can be seen in Figure 3.2 and the receiver in Figure 3.3, where their basic units are composed by the already required processing unit with an extra controller unit and buffer. The processing unit processes data until the buffer reaches the almost full state and resumes it when signalled that buffer is below a certain threshold. The controller block is responsible for signalling the processing unit of such cases while keeping track of buffer fill level.

The blocks of DLL, FEC and modulation were implemented in the FPGA taking into account the asynchronous architecture. Each of these blocks ensure its buffer fill level and signal the adjacent blocks to prevent full or empty buffer states. An important part of modulation is the need of an analogue signal to be made real and positive to ensure the LED requirements. FEC codes are another value added feature to the project due to many

variables in VLC channel that can compromise performance. VLC systems are very sensitive to noise and interferences and these can degrade system performance. This signal degradation can arise due to many sources and these will be fully discussed in section 3.2 together with FEC code techniques to overcome them.

Now that the reader has an overview of VLCLighting architecture and the importance of some of its blocks, it is time to understand the importance of the proposed work on a reliable DLL protocol.

3.1 Data Link Layer concepts

The OSI reference model is a seven-layer conceptual model developed by the International Standardization Organization (ISO) that describes the standards for inter-computer communication. The purpose of the OSI reference model is to guide vendors and developers in the development of digital communication products and software programs, as well as, facilitate clear comparisons among different communications tools. OSI reference model representation can be seen in figure 3.4. In order to understand the importance of this work, a full description of its layers will be performed.

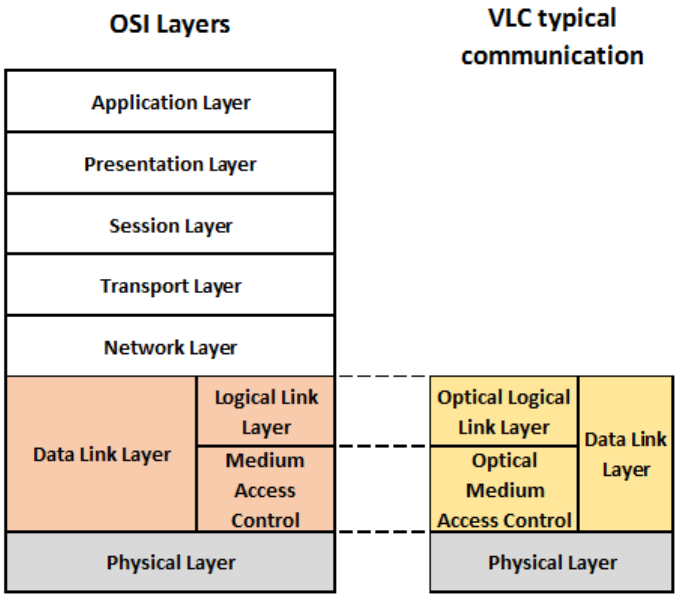

Figure 3.4: OSI VLC adaptation

The main concept of OSI model is that the process of communication between two endpoints in a telecommunication network can be divided into seven distinct groups of related functions called layers. So, in a given message between users, there will be a flow of data down through the layers in the source device, across the network and then up through the layers in the receiving device. At each level, two entities at the communicating devices exchange protocol data units (PDUs) by means of a layer protocol. Each PDU contains a payload, called the service data unit (SDU), along with protocol-related headers and/or footers. The

seven OSI layers are stacked from Application Layer to Physical Layer and they perform the communication functions of a telecommunication or computing system without regarding their underlying internal structure as well as their technology. Another model and set of communication protocols used on the Internet and similar computer networks is the Internet protocol suite, also know as the TCP/IP protocol. TCP/IP provides end-to-end connectivity specifying how data should be packaged, addressed, transmitted, routed and received at the destination. It is organized into four abstraction layers called Application, Transport, Internet and Link. This DLL study is contemplated in both models as DLL in OSI model and Link layer in TCP/IP protocol. Since TCP/IP model lacks the formalism of the OSI model, the OSI layer will be the one on the focus of the discussion.

Starting from the top of OSI model we have the Application layer that interfaces with network applications like mail, web and file transfer. The next layer is Presentation Layer that provides a context for communication between layers, like encryption, decryption and compression. This layer prepares the data from the lower layers for "presentation" to the application layer. Session layer establishes and maintains communication link. It controls the dialogues between computers, where typical protocol examples are handshaking protocols. It also controls duplexing, termination and restarts of the communication. It is important to remark that these three layers are seen by many as an whole and their datagrams are called Upper Layer Data. The Transport layer produces transparent transfer of data and the usual protocols operated here are the TCP for reliable connections and UDP for unreliable ones. TCP is used to make sure that the datagrams arrive at the destination and UDP is used when latency is a more serious issue. The Transport layer provides end-to-end connections, reliability and flow control. The datagrams of this layer are called Segments. The next layer is called Network layer and it is where routers operate. Network layer provides connections between hosts and different networks with logical addressing protocols like IPv4 and IPv6. As previously mentioned, it is in this layer that routers operate so, it is where the routing of packets takes place. The DLL that will be the focus of this work adds physical addressing by providing connections between hosts on the same network. DLL datagrams are called Frames. The last layer is called Physical layer, it describes electrical and physical specifications of the used devices. The datagrams in this layer are called symbols. The communication according to the OSI model is achieved when a datagram goes down through each one of the mentioned layers. Upper layer data is sent down from the application layer and once it arrives the transport layer, it is encapsulated into a segment, so the segment header is attached to the upper layer data. Then, it goes down to Network layer where the segment is encapsulated into a packet. At the DLL the packet is encapsulated into a frame and at Physical layer it is converted into symbols that are sent out to the network. The receiving process is made backwards, it goes up through the different layers. In each lower layer the encapsulation is stripped away, leaving the Upper Layer Data at the end of the process.

The DLL provides the building blocks for communication across a variety of physical media. It connects upper-layer processes to the Physical layer, in other words, it places data on and receives data from the network layer and then provides data deliver between services. The DLL is divided into two sub-layers called MAC and LLC. The Network layer packets do not have a way to cross the physical media. It is the role of the DLL to format these data packets into frames and send them out to the network. Typically, the DLL is also responsible for fragmentation, re-assembly, error handling, addressing and medium access control. The DLL is the lowest layer in the OSI model that is concerned with addressing: labelling information with a particular destination location. Each device on a network has a

unique number, usually called MAC address, that is used by the DLL protocol to ensure that data intended for a specific device gets to it properly.

In order for data to be sent across the network, the Network layer packets have to be encapsulated and formatted into frames. The data packet is taken from the network layer above and fitted with a header and trailer in order to be sent through the physical layer. This process is know as encapsulation. The header contains information that is specific to the type of network and the protocol being used Typical components of the header are the start frame, address field, which contain the destination and source address of the frame. It also includes a type/length field that is used to characterize the type of data being sent or frame length. The trailer comes at the end of the frame and contains two parts: the frame check sequence (FCS) and the stop frame. The stop frame is optional in the the case of length field usage. The FCS field is used for error correction where the sender creates a verification code based on the package content and places it in this field. When the frame arrives at its destination the receiver makes its own calculations and compares them to the ones placed in FCS field. Once the same calculations are made at the receiver, the information is disposed in case of FCS field is not a match. This value uniquely represents the data contained in the frame and is used by the receiving node to detect bit errors occurring in the physical link between nodes.

The DLL is conceptually divided into two sub-layers called LLC and MAC. These sub-layers provide full functionality for different protocols and hardware. The LLC sub-layer prepares data for transmission by giving specific information regarding which network protocol is to be used in the frame. It allows a variety of protocols to be used in the same hardware. The successful transmission of a DLL frame requires that the frame is received, intact, by the receiver, where collisions can cause undesired corruption to them. In order to minimize the loss or corruption, medium access control protocols are used by the MAC sub-layer that regulates the placement of data on the physical medium, controlling the access to the physical hardware. It also provides specific information that allows the frame to meet the specifications of the physical device where data is being sent across. This sub-layer also marks the beginning and end of a frame. Typical DLL protocols are chosen based on network hardware implementation. Some examples are HDLC (High level Data Link Control), PPP (Point-to-Point Protocol) and the most widely known Ethernet.

VLC systems are systems where data is transmitted wirelessly over short distances and where DLL protocol is in general a key issue for performing hard real-time communications. If this layer is not able to give tight time and reliability guarantees (medium access time), this can be hardly corrected by other protocol layers. A comparison between the conceptual OSI model and the usually implemented layers for VLC systems can be seen in figure 3.4. It is important to notice that VLC systems usually do not consider logical addressing and implement only the two lower OSI equivalent layers. As previously said, OMEGA project and DVB-RCT supported and inspired this work. The proposed DLL structure follows this inspiration lines. The first one, with its OWMAC Specification, provided insight into a DLL structure for VLC, while the second one provided some insight into broadcast MAC specifications.

OMEGA OWMAC specification that can be seen in [46] was a key-part that provided a detailed insight in a MAC and LLC protocol for IRC and VLC. The MAC service in the OMEGA architecture is fully distributed, meaning that all devices provide all required MAC functions and no device acts as a central coordinator. The coordination between the devices within optical range is achieved by the exchange of beacon frames. This periodic beacon transmission enables device discovery, supports dynamic network organization, and

provides support for mobility. The basic timing for the network and scheduling information for accessing the medium are also performed with these beacons. The OMEGA basic timing structure for frame exchange is a superframe that starts with a Beacon Period (BP). Each device protects itself and its neighbours BPs for exclusive use of the beacon protocol, ensuring that no transmissions other than beacons are attempted during the BP of any device. MAC beacon frames are intended to be received and interpreted by all devices and hence their frame payloads are transmitted with the lowest available data rate in order to be decoded by all recipients. Other frames are exchanged in a more restricted context and their frame payloads are transmitted at the higher possible data rates. This information is included in OMEGA MAC Headers. Fragmentation of data frames are also performed in OMEGA project to reduce the data loss in marginal links, as well as an implementation of acknowledgement policy that assures the delivery of a frame. If the source does not receive the requested acknowledgement then it may retransmit the frame or discard it. OMEGA also implemented a Multiplexing sublayer to enable the coexistence of concurrently active higher layer protocols within a single device, routing outgoing and incoming data to and from their corresponding higher layers. OMEGA MAC frame which consists of a fixed-length MAC Header and an MAC Frame body with an optional variable length is depicted in Figure 3.5.

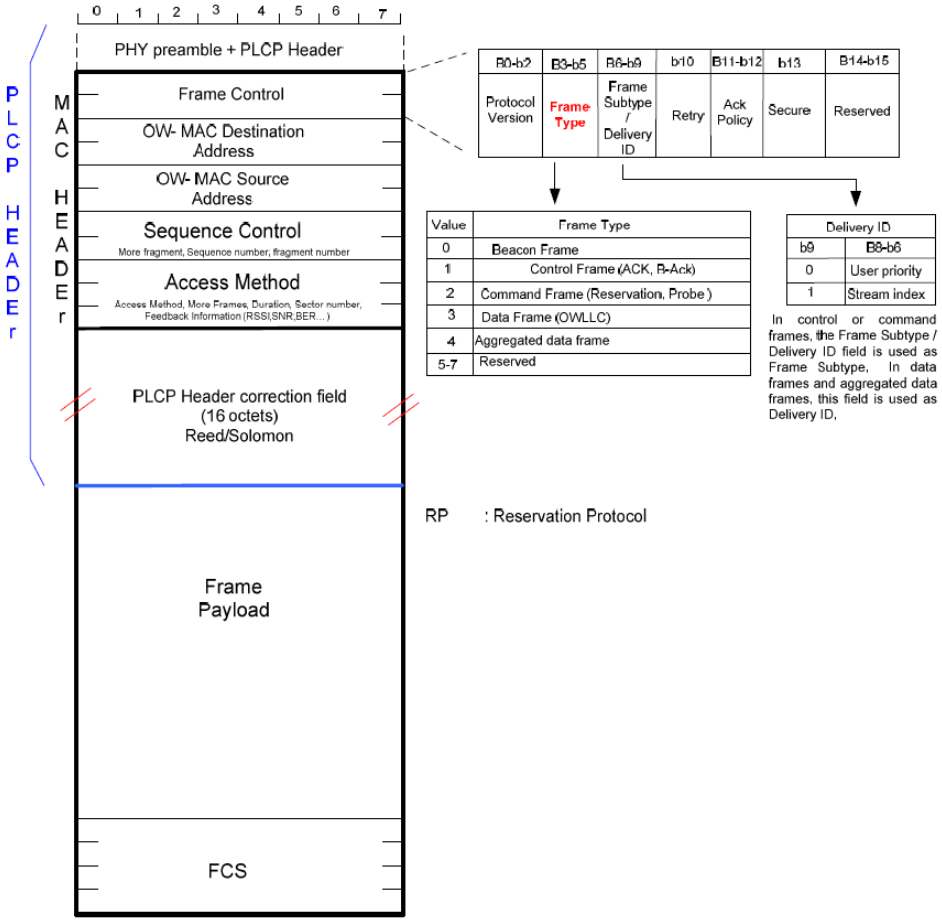

Figure 3.5: OMEGA General MAC Frame format [46]

The Frame Payload in OMEGA has a length range from zero to 4095 Bytes. It carries the information to be transferred to a device or group of devices. If the payload length is zero, the FCS is not included and there is no MAC Frame Body. The MAC header is composed by a Frame Control Sequence, MAC Destination and Source addresses, Sequence Control, Access Method and PLCP Header correction field.

The Frame Control consists of a Protocol version (3 bits) used for device supporting checking; a Frame Type (3 bits), a Frame Subtype (4 bits) to assist the receiver in the proper processing of the frame; a retry bit that identifies a retransmission of frame; an acknowledge policy parameter (2 bits) set by the transmitter; a secure bit to identify the usage of a temporal key; and 2 reserved bits. In OMEGA project, five frame types were planned but only three were implemented, which were the Beacon, Control and Data frame types. Beacon frames are used to determine the expected reception time at the receiving device as well as to estimate the impact on superframe synchronization and increment its start time accordingly. Control frames are used for Acknowledgement purposes Data frames, as the name, suggests to send the data.

The MAC Destination and Source addresses have a 16 bits size each. They identify the source device and the destination device. A particular case of destination address is the 0xFFFF used for broadcast. The Sequence Control identifies the order of the data units and their fragments. It is composed by one fragment bit, sequence number (12 bits) and a fragment number (2 bits). The fragment number is zero in the first or only fragment and is incremented by one for each successive fragment of that service data unit. The sequence number is used for duplicate detection of frames sent using the acknowledgement policy. For this purpose a dedicated counter for different source devices, destination devices and fragments is assigned. The More Fragments field is set to zero to indicate that the current fragment is the sole or final fragment of the current service data unit, otherwise is set to one.

Access Method parameter is divided into Duration, More Frames, Access Method and Feedback information. The Duration field is 13 bits in length and is set to an expected medium busy interval after the end of the PLCP header of the current frame in units of microseconds. Its value is used to update the network allocation vector. More Frames is set to zero if the transmitter will not send further frames to the same recipient during the current superframe. Access method sub-parameter must be set for each packet and must be updated for each time slot. It is set differently depending on the reservation type (half-duplex or full-duplex) for each packet sent in the given time slots. The Feedback information field (8 bits) is used by the receiver of the frame to assess the channel condition that could be based on the Received Signal Strength Indication (RSSI), BER and Packets Loss.

Now that the reader has an overview on OMEGA OWMAC specification, DVB-RCT contributions to this work study will be presented. In order to provide asymmetric interactive services supporting broadcast to the home with a return channel, a MAC standardization was conducted and the frame structure can be seen in Figure 3.6.

DVB-RCT general MAC format has several common frame characteristics with OMEGA OWMAC. As OMEGA, it also has a Protocol Version parameter with three bits to identify current MAC version. Syntax Indicator (2 bits) indicates the addressing type contained in the MAC message and together with Protocol Version make the Message_Configuration parameter. DVB-RCT Message_Length indicates the length of the MAC message and it is present only in downstream messages. MAC address parameter identifies the MAC address of the Network Interface Unit (interface between the downstream and upstream channel). Another common parameter to OMEGA is the Fragment_Count field that identifies the fragment in a

MAC message when multiple fragments are transmitted. The last DVB-RCT MAC parameter is the one that contains the data and it is called MAC Information Elements.

MAC_message(){	Bits	Bytes	Bit Number/Description
Message_Length	16	2	In DS messages only.
Message_Configuration		1	
Reserved	2		7..6
Protocol_Version	3		5..3
Syntax_Indicator	3		2..0
Message_Type	8	1	
If (Syntax_Indicator == 01 || Syntax_Indicator == 10) {			
MAC_Address	(48)	(6)	
}			
If (Syntax_Indicator == 10) {			
Fragment_Count	(8)	(1)	
}			
MAC_Information_Elements ()		N	
CRC	8	1	In DS messages only.
}			

Figure 3.6: DVB-RCT MAC message structure [47]

This section provided the reader with the basic concepts that were used to design and implement the DLL layer for VLCLighting project. In the next section, project constraints will be discussed.

3.2 Project constraints

In order to achieve a compromise between the proposed DLL and its adjacent block (FEC), a study of their efficiencies impact in each other is required. Therefore, this section will present the study conducted to achieve the best DLL frame scenario and FEC code to be used. First, a brief introduction to system parameters and their definitions will be discussed. The performance analysis for non-binary block codes over memoryless channel was a key part of this study. In order to evaluate system performance in different conditions, the probability of bit error and Signal-to-Noise Ratio (SNR) are two of the most important performance parameters to taken into account. One of the most important ways to determine the quality of a digital transmission system is to measure its Bit Error Rate (BER). As the name implies, BER parameter is the number of incorrectly received bits divided by the total number of transferred bits during a time interval. The definition of BER can be translated into the formula 3.1 [48]:

$$BER = \frac{Number\,of\,errors}{Total\,number\,of\,bits\,sent} \tag{3.1}$$

The SNR is a measure of signal strength relative to background noise. This is a measurement of prime importance in all applications from simple broadcast receivers to those used in cellular or wireless communications and it is expressed by the equation 3.2.

$$SNR = \frac{P_{signal}}{P_{noise}} \tag{3.2}$$

The SNR is more usual to see expressed in a logarithmic basis using decibels like in 3.3:

$$SNR_{dB} = 10 \times \log_{10}\left(\frac{P_{signal}}{P_{noise}}\right) \tag{3.3}$$

If the medium between the transmitter and receiver is good and the signal to noise ratio is high, the BER will be very small. However if noise can be detected, there is a chance that the BER will need to be considered and its impact studied.

VLC system's reliability can be compromised by noise and interference from several sources. These systems need to be able to cope with ambient light like sun and moon, and existing artificial light sources. Since this VLC system aims at public lighting, atmospheric attenuation is a big issue in outdoor scenarios because it cannot be conveniently controlled and can cause severe signal degradation. One of the most compromising phenomenons is fog. When light propagates through it, the electromagnetic waves interact with the water droplets suspended in the air. This interaction causes the light to attenuate and scatter. Light attenuation in a medium occurs due to absorption and scattering but due to fog size particles light attenuation is negligible and only the scatter effect is considered. Scatter effect varies with the droplet size and their concentration. Some studies are being conducted to provide intelligent lighting systems with the dynamic changing of the photometric characteristics of luminaires and the luminance of the road surface depending on weather conditions [49]. Isotropic scattering which occurs when outgoing light scatters into every direction with equal likehood depends on the wavelength of the light, where shorter wavelengths (blue,violet) scatter more, loosing light directionality. Channel blockage is other challenge to overcome. Shadowing, as well as, lost of LOS are some of its examples. The last error source is multipath that is more present in indoor scenarios and is due to reflections and signal time dispersion. A particular issue raised by this last one is InterSymbol Interference (ISI) that occurs when one symbol interferes with subsequent symbols.

All these can contribute to errors that can be characterized in two types: random and burst errors. The random and burst errors induced by these phenomenons must be handled in order to guarantee a minimum system performance. When errors occur in many consecutive bits rather than occurring independently of each other, methods of burst error correction are employed. Interleaving is a technique that introduces more robustness with respect to burst errors. It is used to improve the performance of the forward error correcting codes. This technique shuffles source symbols across several code words transforming the burst error into a more uniform distribution of errors. In the VLCLightning project, FEC techniques such as ReedSolomon (RS) and/or convolutional codes provide the receiver error detection and correction capabilities. RS codes have both random and burst errors correction capabilities, while convolutional codes handle random errors better at the cost of higher implementation complexity and larger code words for the same performance.

To find the sweet spot to achieve optimal values for the project, as previously said, a compromise between FEC and the DLL frame size has to be accomplished. Performance equations of the DLL frame can be seen in 3.4 and RS efficiency calculation is achieved through the use of 3.6.

$$Frame\ efficiency = \frac{Payload\,size}{Frame\,size} \tag{3.4}$$

$$Frame\ size = Header\,size + Payload\,size \tag{3.5}$$

$$Reed\ Solomon\ efficiency = \frac{K}{N} \tag{3.6}$$

Prior to this efficiency study a total of 64bits (8 Bytes) were identified as required to all header information and with this value in mind, different frame efficiencies are obtained with payload size variation (Equation 3.5). The DLL frame efficiency is given by the payload size divided by the total size of the frame, as it can be seen in 3.4. On the other hand, the Reed-Solomon code efficiency is calculated with the information symbols (K) and the length of the non-binary code in elements (N). A systematic (N, K) block code consists of the already mentioned K information symbols and N-K parity check symbols. This N-K extra number of elements guarantee the code correction up to $t = \frac{1}{2}(N - K)$ symbols and can detect up to $N - K$ erroneous elements as shown in the equations 3.8 [50].

$$D_{min} = N - K + 1 \tag{3.7}$$

$$t = \left[\frac{1}{2}(D_{min} - 1)\right] = \left[\frac{1}{2}(N - K)\right] \tag{3.8}$$

Paper [51] provides the basis for the SNR value that will be later discussed. Preliminary results show a 1×10^{-3} BER result with a single Blue LED over 1,75m distance. Since this experiment was conducted with a single LED and the project aims at the exploitation of 255 LED matrix over even further distances, the SNR value is expected to be higher because it has a close relation with the optical power. Aside from this collected data, a study with both convolutional and RS techniques was conducted, showing greater benefit from the usage of the last ones for SNR values higher than 5.8dB (Figure 3.7).

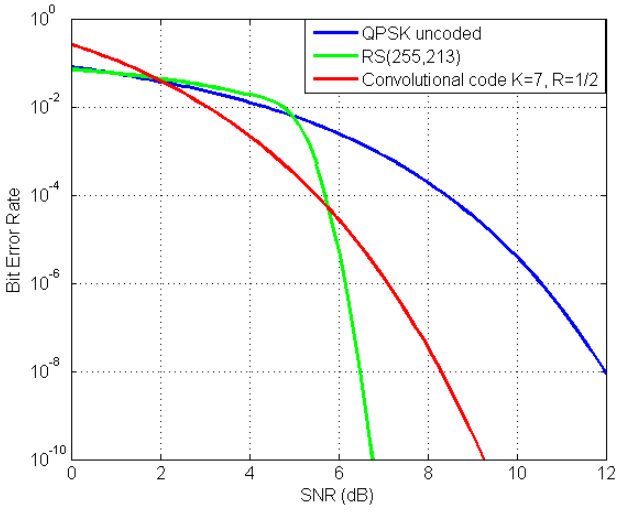

Figure 3.7: Error Correcting codes comparison

Bearing in mind the previous Figure 3.7, the reader can perceive that RS(255,213) code with an efficiency of 83,53% outperforms a convolutional code with only 50% of efficiency after a certain SNR value (5.8dB). Given this efficiency gain, RS codes were the obvious choice and

will be the focus of the present discussion. It is important to remark that RS symbol size is given by: 2^k where k was considered to be 8bits (1Byte) that correspond to the addressing of the majority information also performed in bytes. This matching criteria makes the size of the smaller unit of the FEC codes easier to match the sizing of its adjacent blocks.

With the 8bits k value it is possible to achieve RS N value with Equation 3.9. As stated before, this N represents the length of the non-binary code in elements that multiplied by k gives the total binary size of one codeword (255 Bytes).

$$N = 2^k - 1 \tag{3.9}$$

One reason for the importance of the RS codes is their good distance properties [50]. The minimum distance of non-binary code is denoted by D_{min} and the error-correcting ability of a RS code is determined by its minimum distance as previously stated in Equation 3.8. The minimum distance is given by 3.7.

After FEC technique choice (RS), the next step was to estimate BER value over an Additive White Gaussian Noise (AWGN) memoryless channel with OFDM and QPSK modulation. AWGN is a basic noise model used to simulate the effect of many stochastic processes that usually occur in reality. AWGN is often used as a channel model where a linear addition of wideband or white noise occurs with a constant spectral density and a Gaussian distribution of amplitude. It is important to take into account that this model does not account for non-linearity, dispersion, interference and fading phenomenons.

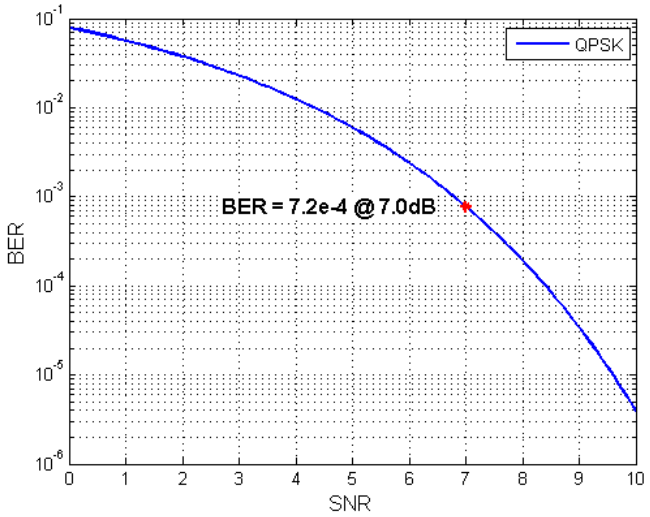

Figure 3.8: QPSK performance over AWGN channel

The BER against SNR curve is presented in Figure 3.8. The value of 7dB SNR was considered because it presents a good approximation to the 1×10^{-3} probability obtained in [51]. As can be seen in Figure 3.8, the 7dB SNR results in a BER of 7.2×10^{-4}. The simulated BER value was conducted for only QPSK modulation because it bases on a channel response approximation. This approach of the channel response considers that the project modulation with OFDM and QPSK can be approximated with a narrowband channel where only the last one was used, therefore a study contemplating only QPSK is considered in this study.

"A nonbinary code is particularly matched to an M-ary modulation technique for transmitting the 2^k possible symbols. Each of the 2^k symbols in the q-ary alphabet is mapped to one of the $M = 2^k$ orthogonal signals. Thus, the transmission of a code word is accomplished by transmitting N orthogonal signals, where each signal is selected from the set of $M = 2^k$ possible signals. The symbol error prabibility P_M and the code parameters are sufficient to characterize the performance of the decoder" [50]. The modulator, the AWGN channel and the demodulator form an equivalent (M-ary) input, discrete (M-ary) output, symmetric memoryless channel, where the channel model can be illustrated as in Figure 3.9.

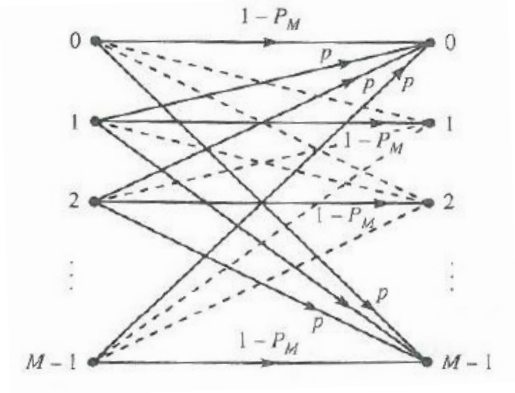

Figure 3.9: M-ary input, M-ary output, symmetric memoryless channel [50]

In figure 3.9 the reader can observe a graphical representation of a symmetric memoryless channel with different probabilities of the correctly reception of a symbol. The p referred in the equation 3.10 is the probability of occurrence of a symbol change. Since P_M is the symbol error probability, $1 - P_M$ is the symbol successfully received probability.

$$p = \frac{P_M}{M-1} \tag{3.10}$$

$$P_M = 1 - \left(1 - Q\left(\sqrt{2R \times SNR}\right)\right)^m \tag{3.11}$$

It is important to mention that all MatLab code elaborated for this efficiency study can be found in Appendix A, and if further questions arise in this section its reading is strongly recommended.

In order to perform a study of BER at the output against different RS codes efficiencies, a set of probabilistic formulas were considered and deduced. As it can be seen in [50], the symbol error probability is given by the equation 3.12. Then, if the symbols are converted to binary digits the bit error probability can be calculated with the equation 3.13 [50].

$$P_{es} = \frac{1}{N} \sum_{i=t+1}^{N} i \binom{N}{i} P_M^i \left(1 - P_M\right)^{N-i} \tag{3.12}$$

$$P_{eb} = \frac{2^{k-1}}{2^k - 1} P_{es} \tag{3.13}$$

However, in this case study the bit error probability was considered to be equal to the symbol error probability. This approach was done because in this study we have really small error probability and it is therefore highly unlike to have more than one incorrect bit per symbol. This small probability does not mean that more than one bit cannot be incorrect but it simplifies the calculation without comprising accuracy.

The simulation in Figure 3.10 assumes several RS(255, K) codes which results in 255 codewords storing K symbols of data. With this graph, the minimum code that is worth choosing could be extrapolated, in other words, it indicates that the right RS code to use is above RS(255,165).

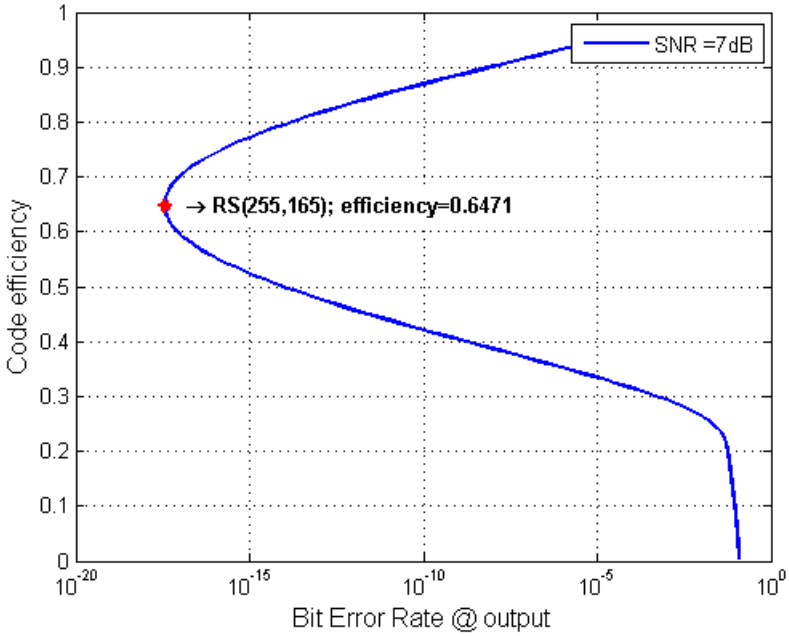

Figure 3.10: Bit Error Rate vs efficiency

Despite the obvious tendency of a decrease in code efficiency with the increase of parity bits (Figure 3.10), the reader can observe that with that compromise, the BER does not go below a certain value. After it, it is not worth the sacrifice in code efficiency. This effect was expected, because since the energy per bit is reduced with stronger RS codes, the SNR will increase. This SNR increase will lead to a bigger error probability as can see in Equation 3.11. With this given threshold in mind, simulations with different N sizes in RS codes were made from it (RS(255,165)) up to RS(255,253) with a step of 2.

To choose the appropriate combination between FEC code and DLL frame size, P_{es}, Block Error Rate (BLER) and Frame Error Rate (FER) were estimated in the respective order. Before going into the details of the mathematical statements in equation 3.14 and in equation 3.15 it is important to point out that the equation 3.12 is used to achieve the symbol error probability (P_{es}). This P_{es} corresponds to the sum of different RS codes with different number of information symbols then divided by the number of elements. With simple probabilistic rules, the reader can comprehend the meaning of the BLER and FER formulas. Since this variable (P_{es}) provides information on the probability of having one erroneous symbol, $1 - P_{es}$

gives the probability of having one correctly received symbol. When $(1-P_{es})^{Number\ of\ symbols}$ is calculated the probability of having all correctly received symbols is obtained and then it is subtracted to one which gives the probability of having all incorrectly received symbols. As it can be seen in equation 3.15, the FER calculation has the same logic of the BLER, but instead of using the probability of having one erroneous symbol, it uses the probability of having one erroneous block. It was considered that a codeword is incorrect when at least one incorrect symbol reception occurs.

$$BLER = 1 - (1 - BER)^{Number\ of\ symbols} \tag{3.14}$$

$$FER = 1 - (1 - BLER)^{Number\ of\ blocks} \tag{3.15}$$

In figure 3.11 the result of the different FER versus the total efficiency can be seen, where the last one is the result of the FEC code efficiency times the DLL frame efficiency.

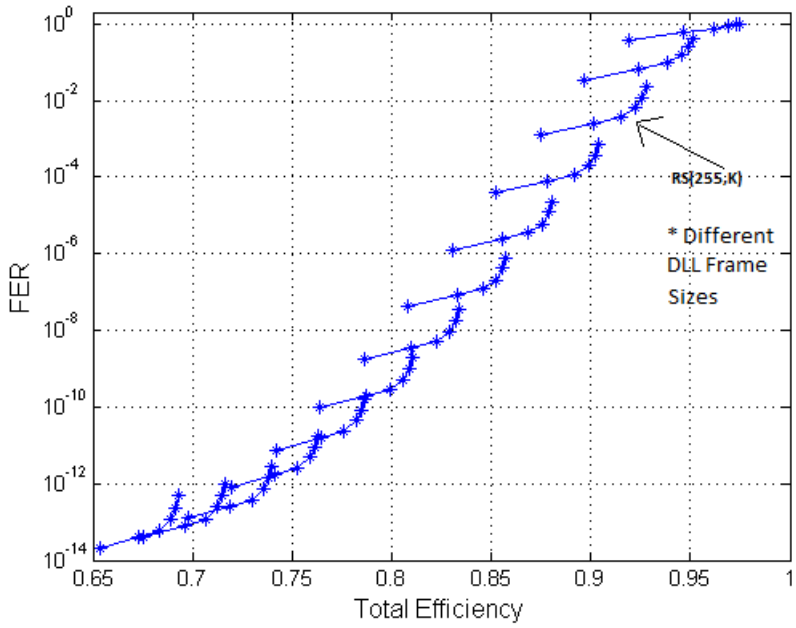

Figure 3.11: Efficiency vs FER Frame

After different FER estimations, the next step is to choose which of the estimated FER is better to meet the system requirements. Since VLCLighting project aims at infotainment broadcast, it was defined a total of 60 minutes without errors as a goal. This error-free time was considered to be ideal to provide the combination of information and entertainment purposes that strongly focus in the video broadcast main objective. FER calculation regarding the system's error free time can be achieved with the help of equation 3.16. The bitrate used in this equation was considered to be $24Mbits/sec$, which was the experimental result at $2m$ transmission over an indoor free space channel with $12MHz$ bandwidth as shown in the paper [51]. During this work, all mentioned frame sizes were in bytes to maintain coherence and in order to use 3.16, it is crucial to convert them to bits by simply multiplying them by 8. The

reader will find that the error-free time should be used in seconds and that with these two last criteria both frame size and time units match the bitrate ($Mbits/sec$).

$$FER = \frac{Frame\,Size \times Time\,without\,error}{Bitrate} \tag{3.16}$$

With all these constraints in mind, the maximum FER to be achieved was calculated to be $3,7926 \times 10^{-7}$ which is obtained with the maximum considered frame size that was 4096Bytes. Each curve in Figure 3.11 represents a different RS code where each asterisk represents a different frame size. Figure 3.11 is a good representation of the impact on FER on the Total efficiency of the code and frame size variation. The maximum FER means that a RS code curve should be chosen below this value.

This approach to choose the FEC code and DLL frame size by only considering the system's total efficiency was proven to be difficult and not ideal. However this study gives useful insight to adapt system frame size and FEC code to different scenario demands and their variation implications. If the frame is too long, it can introduce extra delays to the other requests from the same service. The more bytes transmitted in a frame, the longer each receiver must wait until transmission completes, therefore increasing the packet jitter. It is not worth the effort of having a big frame size where each request filling a small percentage, because only one request from each service will be provided per frame. In other words, shorter bursts of data mean that other requests from the same service got a faster opportunity to transmit, decreasing the delay of received requests. From Figure 3.11 with the increase of frame size there is a gain in total efficiency at a small cost of FER increase. This would increase dramatically the delay between each request and constitute the reason to reconsider this study. So, a step back was taken and a study to understand VLCLighting DLL frame size demands performed.

As previously stated, VLCLighting main goal is video broadcast, so this should be the major concern when choosing the frame size. Video broadcast architectures like the Digital Video Broadcasting - Terrestrial (DVB-T), DVB-T2, Digital Video Broadcasting - Satellite (DVB-S), DVB-S2 and Advanced Television Systems Committee (ATSC) standards were analysed. It was found that DVB-T, DVB-S and ATSC use a standard container format called MPEG Transport Stream (MPEG-TS). Where DVB-T2 and DVB-S2 technologies are an extension of the respective existing standards DVB-T and DVB-S. Besides integrating edge cutting signal processing technologies to allow a better use of the spectral resources, these technologies use a DLL protocol defined by the DVB called Generic Stream Encapsulation (GSE).

GSE concept resides at the same level as MPEG-TS because it provides a compatible broadcast mode for carrying MPEG-TS with the possibility of carrying variable size generic data in base-band frames [52]. The main goal of GSE is to complement MPEG-TS by enabling DVB systems with high flexibility to match the input traffic with the minimum overhead possible. In other words, GSE payload may be encapsulated in a single GSE packet or sliced into fragments and encapsulated in several GSE packets. In VLCLighting project to ensure the lighting functionality, a regular frame needs to be sent with a certain time limit, therefore a variable size frame could compromise it. With this in mind, the framing structure will be based in the MPEG-TS with fixed size frame.

MPEG-TS is a container format for audio, video and metadata transmission. It is clear to understand its formatting purpose of audio and video because it is used in DVB systems. However, its other motive is metadata tranmission, which provides transmission of "data

about data". This metadata has information about the video and audio (such as author, origin and destination locationsn and security restrictions). MPEG-TS packet size is constant (188 Bytes) and it is always divided into a header and a payload. As it can be seen in figure 3.12 the header has a minimum size of 4 Bytes, making the maximum possible payload size to be 184 Bytes.

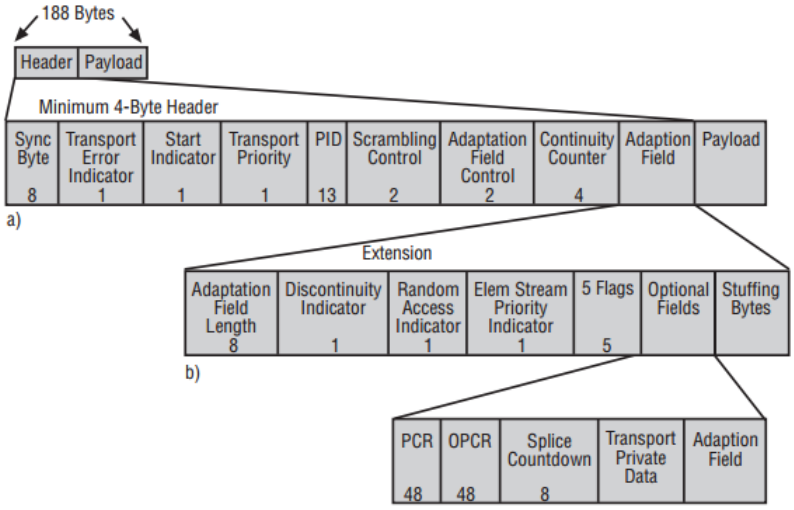

Figure 3.12: Structure of MPEG-TS packet [53]

With MPEG-TS frame size of 188 Bytes in mind and previous study, the choice regarding the DLL frame size should be 256 Bytes. The 256 Bytes value guarantee the adequate means to furnish the HDR service requests and leaves a room of 68 Bytes for the other identified value added service for the VLCLighting project (MDR). The mentioned figure 3.11 shows the result of different 256 frame size with different RS codes that combined with calculated FER of $3,7926 \times 10^{-7}$ gives the basics for system parameters choice. That same information can also be seen in figure 3.13 that has the DLL frame size fixed to 256 and varies the RS code number of data symbols (from 165 to 253). The RS(255,215) is the first RS code to appear below the calculated FER, therefore it will be the one considered in the following discussion.

With all previous discussion concerning the motives for these values selection it was of greater importance to show the reader a more detailed graph of FER versus total efficiency. The following figure 3.13 and 3.14 are the consequence of the restriction imposed in frame size and RS code, respectively. It shows in greater detail the already discussed tendency of both FER and Total efficiency with the variation of RS codes and frame size. It is important to notice that both figures have a small circle that calls attention to the considered value of RS code in the case of figure 3.13 and the considered frame size in the case of figure 3.14.

Both figure 3.13 and 3.14 show that the total system efficiency value of $81,76\%$ could be increased with a sacrifice on the FER and would surpass the limit given by calculated FER with 60 minutes error-free transmission.

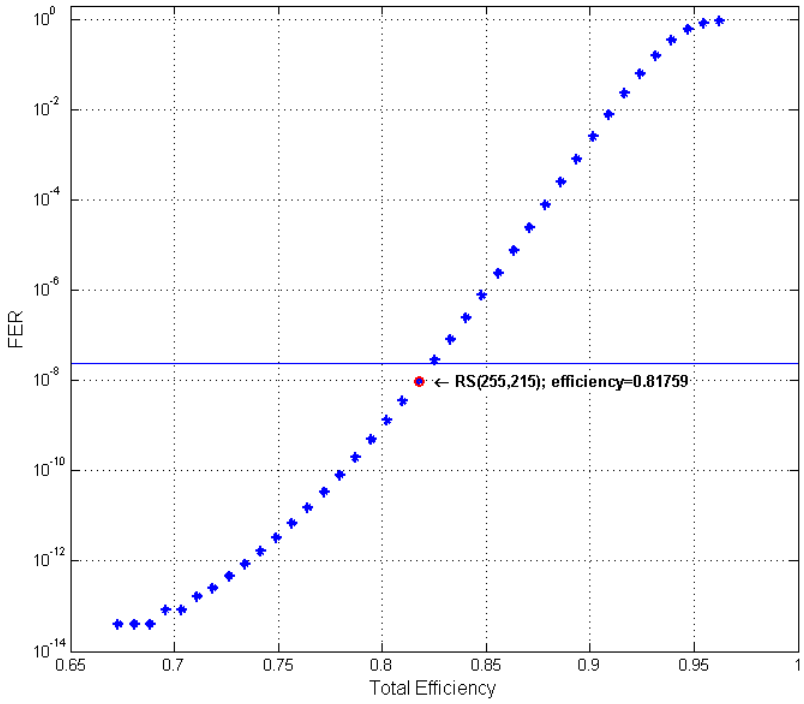

Figure 3.13: Efficiency vs FER for Frame Size = 256

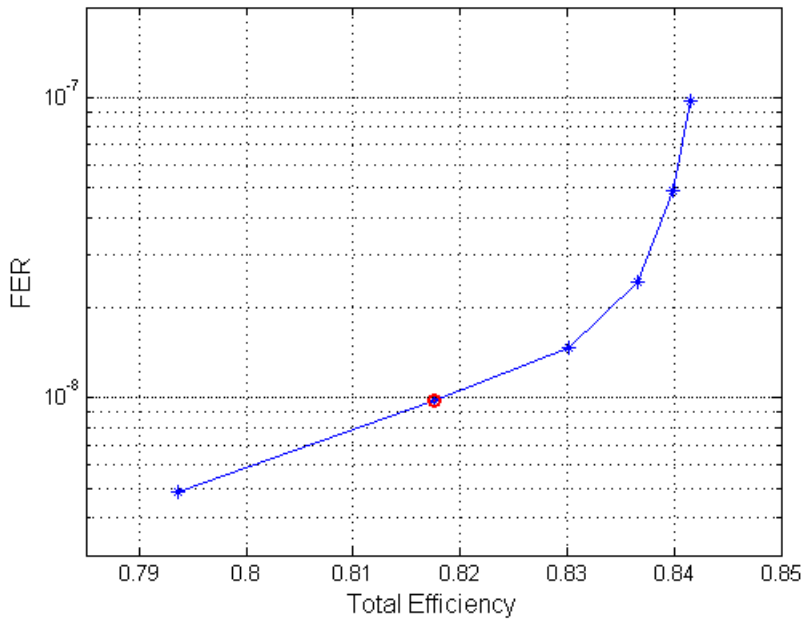

Figure 3.14: Efficiency vs FER for RS(255,215)

With RS(255,215) code fixed and frame size fixed as 256 Bytes another constraint was identified as a consequence of their usage. These results were found to be useful to give an approach required for FEC and frame size selection but small adjustments are required. It is important to regard that with these adjustments MDR size in a DLL frame will be calculated. As previously said, RS(N,K) have a codeword size of N with K information symbols. Taking into account that the RS(255,215) is only able to provide 215 symbols transmission, which are 215 Bytes (each symbol has a size of 8 bits), the DLL frame size of 256 Bytes cannot fit in one codeword. In order to send the 256 Bytes it would be required two RS(255,215) codewords to be used, which would drastically lower the code efficiency (second codeword would only contain 41 Bytes of data). A small adjustment in both was performed with the goal of optimizing the DLL frame to fit in only one codeword. In the light of this objective, the best possible scenario achieved was a RS(255,209) and a DLL frame size of 208 Bytes that provide a total system efficiency of $78,43\%$. However, this means that 1Byte of information will be junk. Nevertheless it was found to be the best compromise scenario. The choice of 208 Bytes was also based on CDMA configuration requirements in memory alignment that will be discussed in greater detail in Chapter 4. CDMA address configuration needs to be 4 or 16 Bytes aligned that compelled this study for the best frame size choice. It was found that 13 CDMA transfers of 16 Bytes was the closest value to 215 Bytes and the already pointed 208 Bytes became the wise choice of frame size. Another crucial improvement needs to be referred. With the goal of improving system's reliability, 32-bit Cyclic Redundancy Check (CRC) will be implemented. This error-detecting code requires 32 parity bits that will transform the FEC code from RS(255,209) to RS(255,213) without changing the frame size. Therefore, the DLL frame size will be considered on this work as 208 Bytes and DLL functionalities will be now discussed in the next section.

3.3 Functionality description

VLCLighting project has identified two types of value added services to be introduced in public lighting systems. A MDR to furnish adequate means for control and management, advertising and infotainment services and a HDR for video broadcast. The broadcast nature of the VLCLighting project requires the use of a DLL to arbitrate the access among these multiple services while enabling flow control and reliability on the transmission. With this goal in mind, OMEGA Sequence Control concept was adapted and introduced in the proposed DLL frame defined in figure 3.15.

Since DLL frame size was set as 208 Bytes in last section and, as will be seen in this section, Header was identified with 8 Bytes, payload size will have 200 Bytes. With the previously identified MPEG-TS frames of 188 Bytes to be sent in the HDR service, MDR service is left with 12 Bytes.

The Sequence Control proposed in this DLL study is intended to fragment both services into the same frame. It is composed by a *More Fragmentation* flag, *Fragment Number* and *Request Size* for both types of services. The Fragmentation Flag is set if the Transmission Frame Data Field holds data that is part of a larger fragmented packet. The *Fragment Number* is set to the number of the fragment within the frame and it is zero in the first or only fragment. *Request Size* is set to the fragment byte size of the service request. Protocol Version was inserted regarding future versions and optimizations on the proposed DLL, it is currently set to 1. It indicates which protocol version is being transmitted and avoids frame

misinterpretations by future outdated devices. During the study of this DLL several header parameters were evaluated, but then abandoned due to project constraints. Since DLL will operate in a broadcast system, different frame types was proven to be unessential. With system unidirectionality we cannot do the medium sensing and therefore evaluate the correct reception and/or its reception time. It was also studied a frame subtype to service prioritization that would avoid starvation of the MDR. Nevertheless, a solution able to split a frame according to each service needs was thought to be a better solution. VLCLighting broadcast characteristics makes MAC Source address and *MAC Destination Address* unnecessary but with localization purposes and user handover warnings in mind, MAC Source address was inserted in the frame. Due to OFDM usage in the project a start and stop beacons were not considered. OFDM already provides frame synchronization because it identifies in the receiving end where the DLL frames start and provides the upper layers with the useful data. *Fragment Number* size was thought to be 12 bits in order to send a maximum service request size of 4096 bytes to leave room for possible future work on a variable payload length. This dynamic length is used in OMEGA project (with a 4095 maximum size), as well as, in many other communication systems. This feature could accommodate higher quality services for VLCLighting project usage. During this work, the frame size was considered to be constant due to existing time restrictions on the data sending, that requires the data to be sent within a maximum period of time. The reader will notice that in figure 3.15 the payload size is set to 200 bytes, as explained in section 3.2.

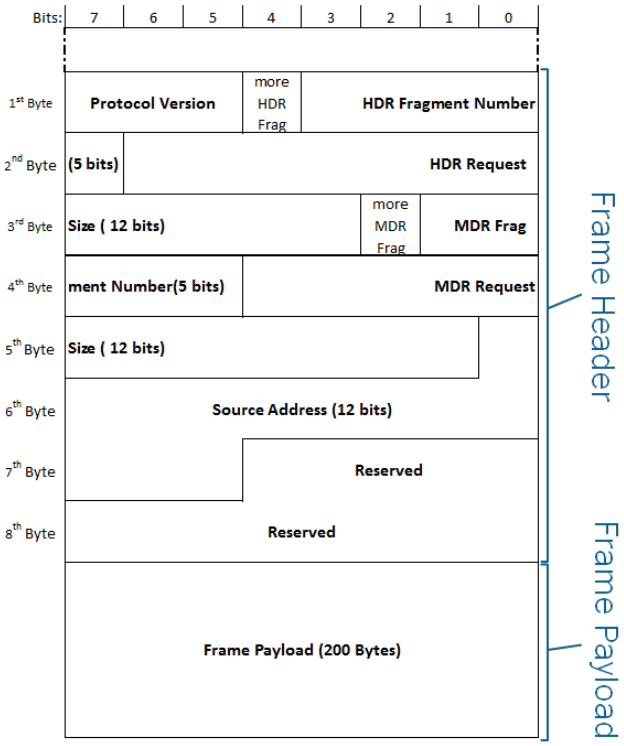

Figure 3.15: DLL frame format

In order to accommodate all the DLL discussed frame parameters, the DLL emitter was structured into five blocks (functions) called Operations Controller, Admission Control, Frag-

mentation Control, Header Coder and Link Management, where the Operations Controller is responsible for managing the remaining blocks (figure 3.16). Admission control, as the name suggests, is responsible for controlling the admission of data. Fragmentation control packages and fragments the data into the so called frames. Header coder is responsible for header calculation and link management for controlling the outgoing data flow. All of these blocks with the exception of Fragmentation Control are connected to a shared memory where the frame is formed before exiting the DLL. On the other hand, the DLL receiver functions are Link Management, Header Decoder, Defragmentation Control, Higher Layer Control and Operations Controller, where the Operations Controller, as in the emitter, is responsible for controlling all the remaining (figure 3.17). The remaining blocks on the receiver are responsible for striping the header and re-assembly the Higher Layer data. All these functions behaviour will be fully detailed in the next section, which will provide the full behaviour of the proposed DLL.

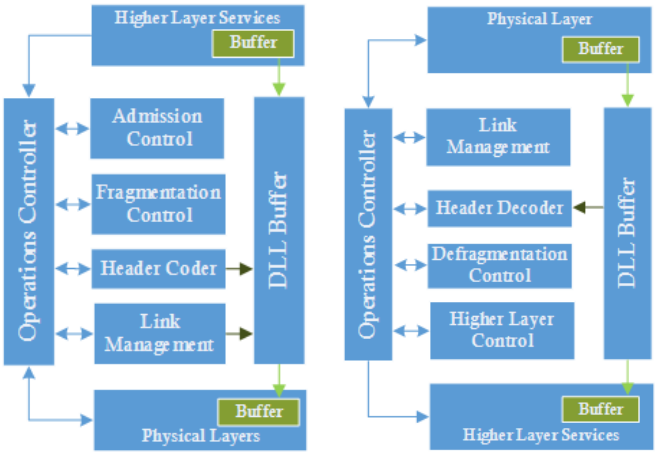

Figure 3.16: Emitter DLL Figure 3.17: Receiver DLL

3.4 Data Link Layer flowcharts

Figure 3.18 shows the proposed DLL emitter that is responsible for structuring the two identified services (MDR and HDR) into frames that are stored in a temporary memory of one frame size (208 Bytes) to then be sent to the different Physical Layers. The Higher Layer signals the DLL with a Service Request Flag and with the *Request Size* that indicates the request data size inserted in the Higher Layer buffers. When Operations Controller receives this information, it signals Fragmentation Control to calculate the byte size of both services to be stored in the DLL internal memory. During this calculation, Fragmentation Control favours the HDR requests that was thought to require 188 Bytes (MPEG-TS payload size) and leaves 12 Bytes for the MDR service. It is very important to regard that these values are not fixed, and are only considered when fragmentation of both services is required. When both services fragmentation is not required, Fragmentation Control tries to maximize payload size usage, making the attendance of service requests flexible. After Fragmentation Control, the Admission Control transfers the data from the Higher Layer buffers to DLL internal memory, and the Header calculates and inserts the header information at the same time. Once the

Operations Controller is signalled that all data was transferred, meaning that the frame is ready to exit the DLL, the Link Management inserts the Source Address and sends it to the proper Physical Layer.

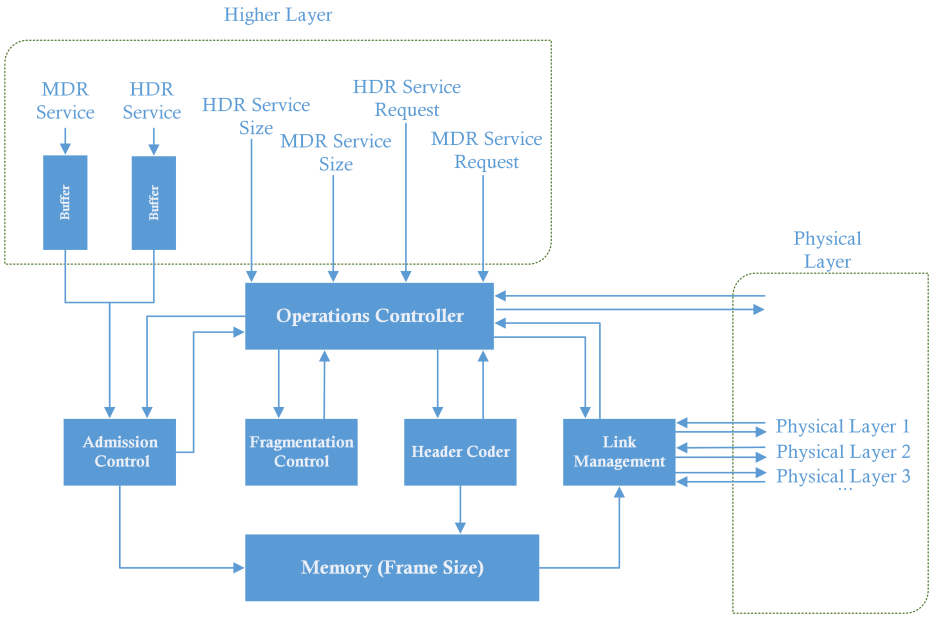

Figure 3.18: Emitter DLL

Bearing in mind the discussed asynchronous architecture of the VLCLighting project, the DLL needs to provide and obey a certain set of signals to be integrated between the Higher Layer and the FEC block. While keeping track of its buffer level, the DLL emitter needs to signal FEC block of the Almost Empty (AE) state, which indicates that the next block must stop processing. This flag ensures that the next block buffer is never empty controlling the flow of data between both. To guarantee the proper functioning of DLL, a flag from FEC is required to signal when a frame from its buffer has been removed. This is required to update all memory addressing parameters and perform the correct address writing on the FEC block buffer. The flowchart of the DLL emitter Operations Controller is depicted in figure 3.19. Since it was established that the maximum number of requests is 32, a buffer for the HDR and MDR request sizes was created with 32 positions. The AE flag is initialized with 1, because no information was sent yet. It was established a minimum FEC buffer level of one DLL frame size, meaning that the AE flag will be set to 0 after one DLL frame is sent. The flag responsible for the Almost Full state is initialized with 0, due to DLL buffer being empty at the beginning.

The first operation to be performed by the Operations Controller is to check the FEC buffer level and see if it is Almost Full. If the condition is true, then DLL must stop processing and wait for FEC block to empty the buffer below a certain threshold that was defined as one DLL frame size. The second task is to ensure that FEC block receives a frame at a certain rhythm. If the measured time is superior to the defined maximum time, a junk frame needs to be sent. After ensuring these operations, Fragmentation Control function is called to calculate the HDR and MDR data size to be sent. Then, it gives the HDR data size , if a HDR request was made, to the Admission Control. Admission Control function is called to configure the DMA

parameters to transfer the HDR data from Higher Level buffer to DLL buffer, then High Level buffer state parameters are updated. After this, Header Coder function is called to calculate and transfer the header information to the DLL buffer. If a MDR request was performed the Operations Controller will wait for the DMA transfer to finish the HDR request transfer and to configure the DMA with the MDR request information. After Admission Control completion, Link Management function is called to insert the Source Address and configure the DMA for the DLL frame transfer to the FEC block buffer.

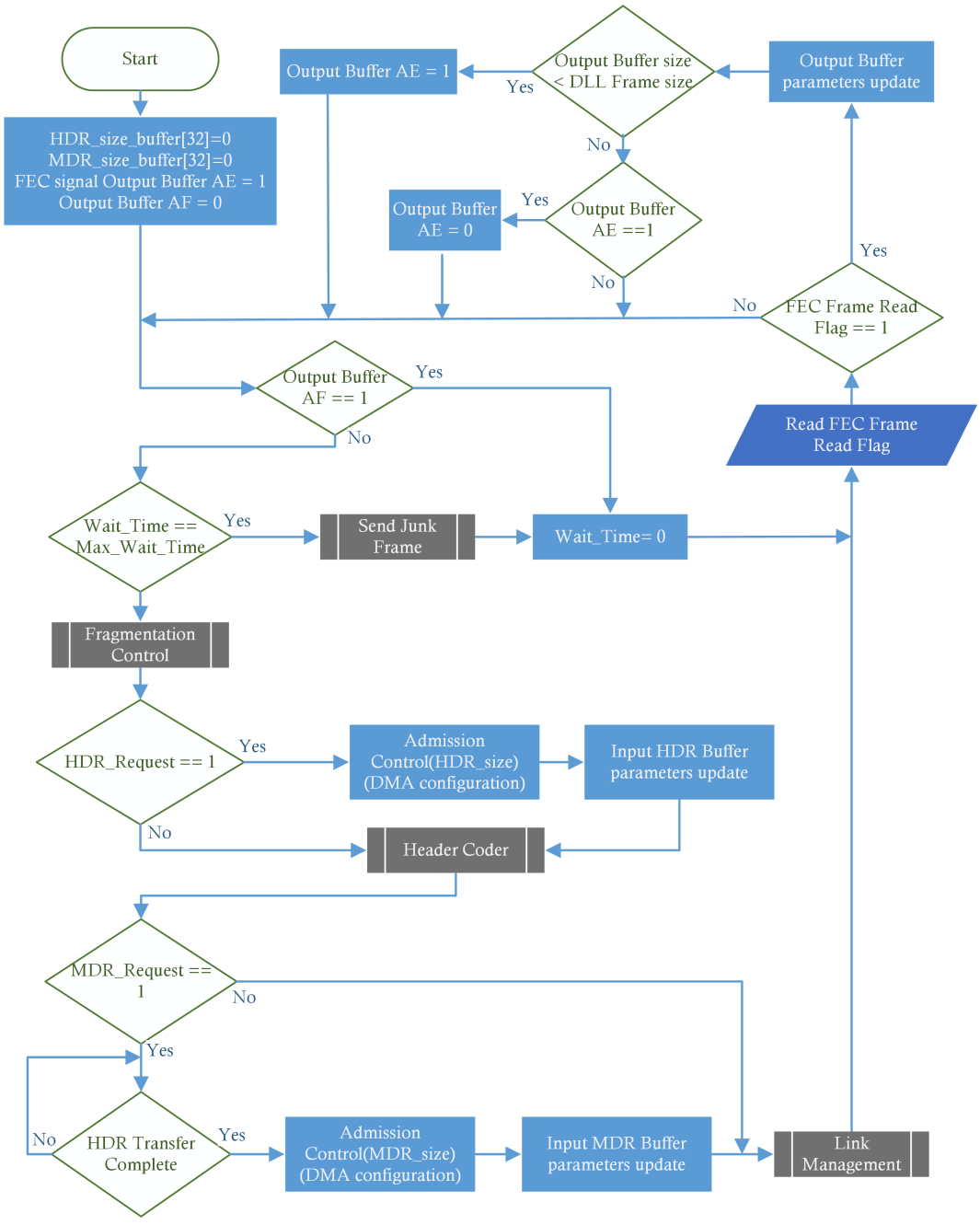

Figure 3.19: Emitter Operations Controller

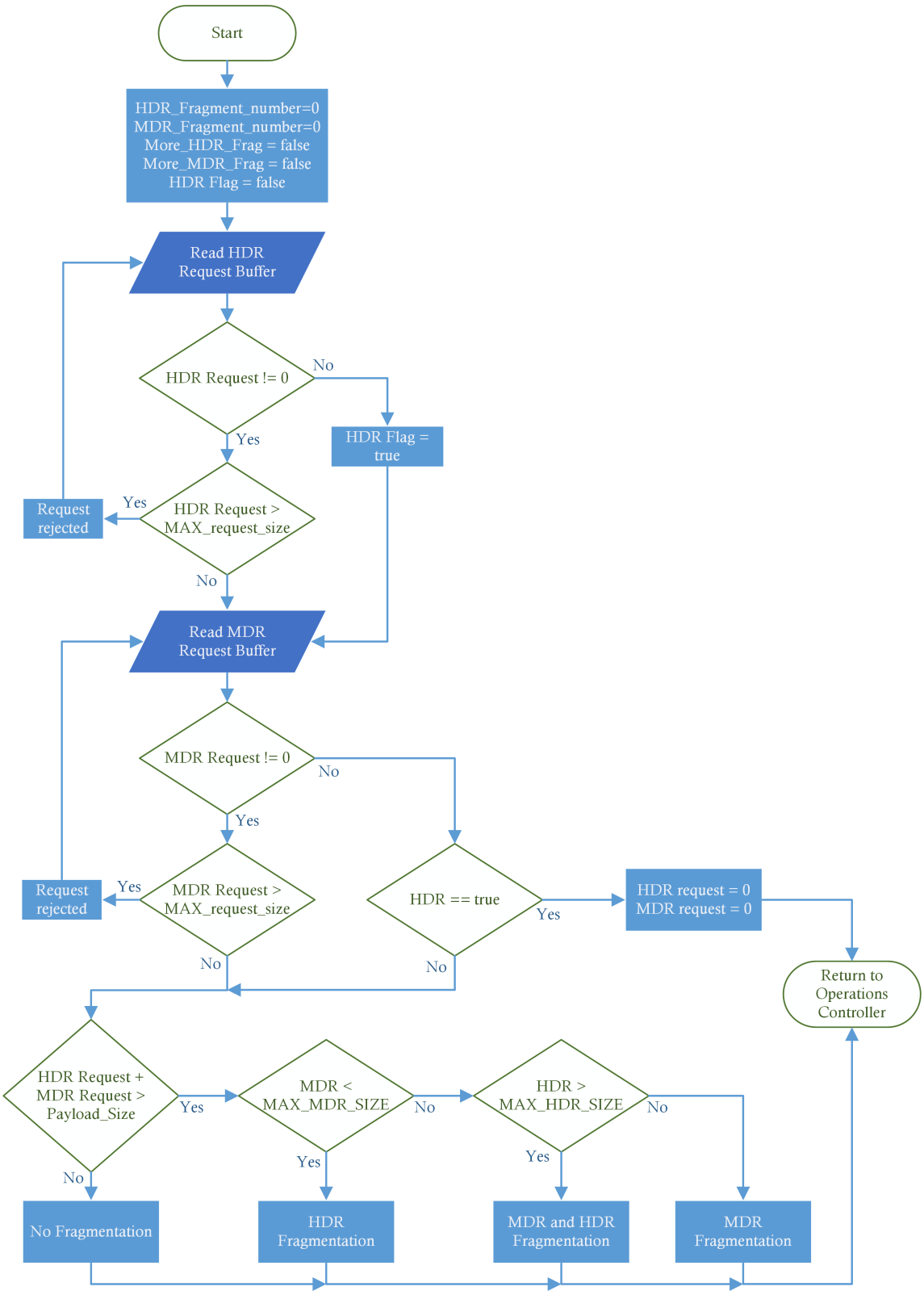

Figure 3.20: Emitter Fragmentation Controller

The Framentation Control function behaviour can be seen in figure 3.20. Fragmentation Control is responsible for providing the Operations Controller with the fragment number calculation and the *More Fragment* parameter for both services. When Fragmentation Control function is called for the first time, fragment numbers are initialized with 0 and *More Fragment* flags are initialized as false. An additional flag called HDR will be required, initiated as false. The first operation of this layer is to read the buffer created by the Operations Controller that contains all request sizes. If a HDR request is available, its value is read. Due to *Fragment Number* frame parameter size of 5 bits, a maximum of 32 fragments can be performed which limits the maximum request size to 6400 Bytes ($32 fragments \times 200 Bytes(FrameSize)$). The request size read is compared to this maximum request size and if the request is bigger than the allowed it is discarded and a new request is fetched from the buffer. If no more HDR requests exist in the buffer, the HDR auxiliary flag is set to one. After, the MDR request buffer level is checked and if a request exists its size is compared with the maximum allowed request size. With these informations, the Fragmentation Control block is able to calculate the HDR and MDR data size to be sent. This calculation is different depending on the five scenarios shown in the flowchart. If HDR and MDR request size is zero, no data will be sent. If the sum of both requests does not surpass the frame size no fragmentation is required and the data to be sent is given by the request sizes. If the sum of both requests surpasses the frame size and the MDR request is lower than a given maximum (16 Bytes) only HDR fragmentation occurs, which means that the MDR data size is given by the MDR request size. The HDR data size is given by the $frame\ size - MDRrequestsize$, the HDR remainder request size is saved. If the sum of both requests surpasses the frame size and the HDR is lower than a given maximum (188 Bytes) both HDR and MDR fragmentation occurs, which implies that the data to be sent is given by the previous maximums and their request remainders are saved. Last case is when the requests sum overpasses the frame size and the HDR is lower than the said maximum, which implies the MDR fragmentation. Once the calculation is performed, the values are returned to the Operations Controller that will resume his processing.

The Admission Control is the following function, its flowchart is shown in figure 3.21. Since DMA configuration requires memory and transfer size alignment, the Admission control first step is to assure that the transfer size is aligned. It does not need to align memory addresses because the Operations Controller already perform those. The next step is to check whether or not the DMA is busy with an concurrent transfer. If DMA is busy, the Admission Control waits for its finish and then configures with parameters given by the Operations Controller.

The Header Coder flowchart is not present in this section because it only has calculations to perform and those are not relevant to enumerate in a flowchart diagram. The Header Coder performs the bitwise operations to fill correctly all header fields that were previously discussed and stores them in the DLL temporary memory.

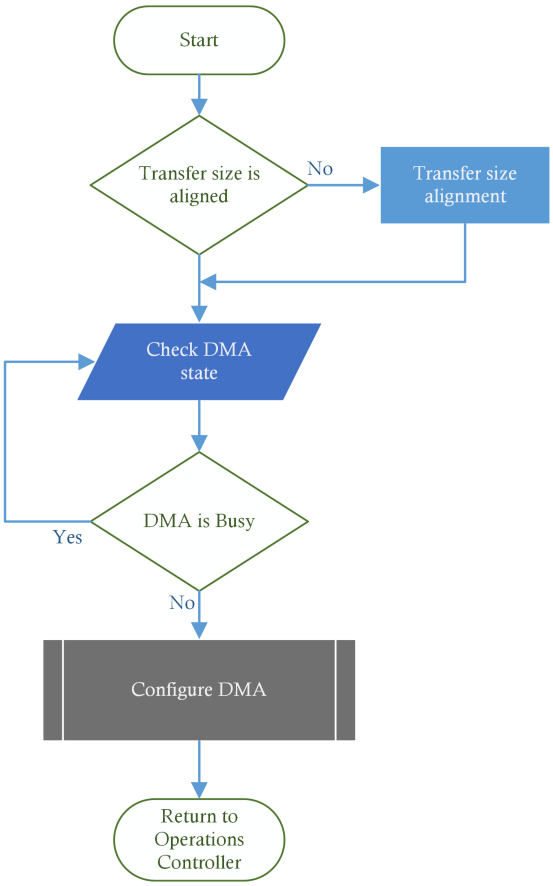

Figure 3.21: Emitter Admission Controller

The last DLL emitter block is the Link Management. Its behaviour can be seen in figure 3.22. The Link Management contains a buffer with all physical addresses and uses that buffer to fill the Source Address frame parameter. The Link Management initializes with 0 an auxiliary variable called *iteration*, and will increase its number while sending the DLL frame to different Physical Layers. The first task to perform is to check if the iteration variable is lower than the number of Physical Layers. If this condition is true, the Physical Address will be fetched from its buffer and inserted in the DLL buffer, the iteration variable incremented. Then, like the Admission Control, the Link Management needs to wait for DMA completion to configure it. This DMA configuration is performed by the Link Management as many times as the number of Physical Layers. Once finished, it returns to Operations Controller with information regarding the transfers.

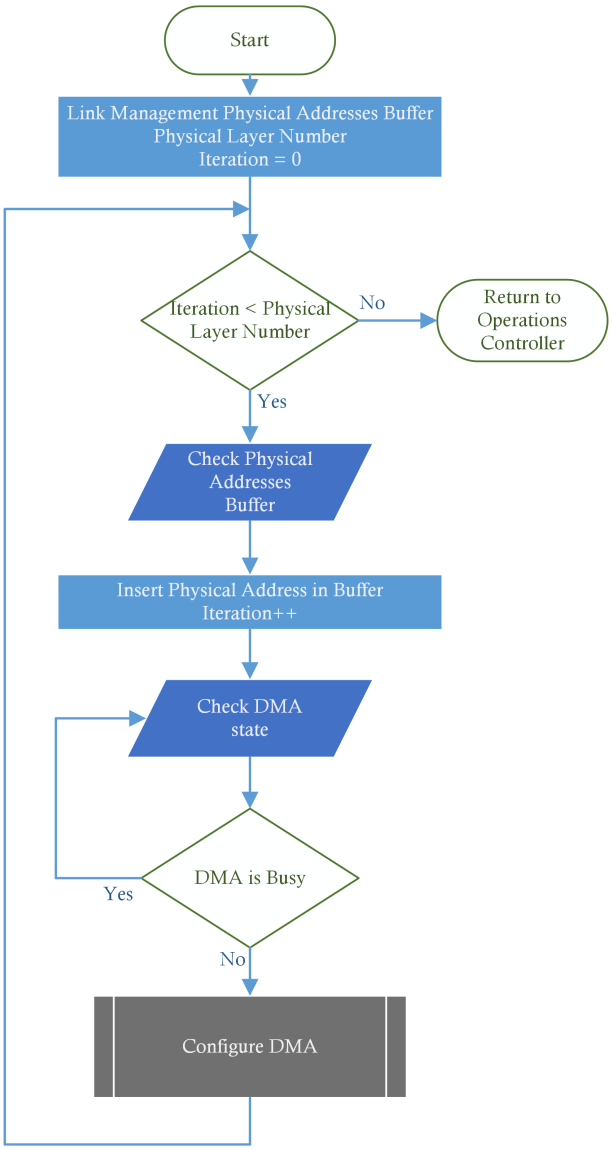

Figure 3.22: Emitter Link Management

During the elaboration of the Receiver DLL functions, similar but inverse reasoning of the Emitter DLL was used. The Receiver DLL structure can be seen in figure 3.26.

The Receiver DLL is signalled by the Physical Layer when a frame is in its buffers. After this signal arrival, the Operations Controller block calls the Link Management function to transfer the data from FEC block buffer to the DLL buffer. As soon as transfer finishes, the Header Decoder is called to evaluate the received frame header. It communicates the HDR and MDR data size present in the frame payload. It is also the Header Decoder function that establishes if the frame was correctly received. Then, Operations Controller calls Defragmentation Control to calculate the data re-assemble parameters to the Higher Layer buffers and check if the received fragment is the one expected. After, the Higher Layer Control is called to transfer the data from the DLL buffer to the corresponding Higher Layer buffer. As soon as the Operations Controller has the information that a Higher Layer request has been

completed, it signals the corresponding Service that a complete received request is present in its buffer.

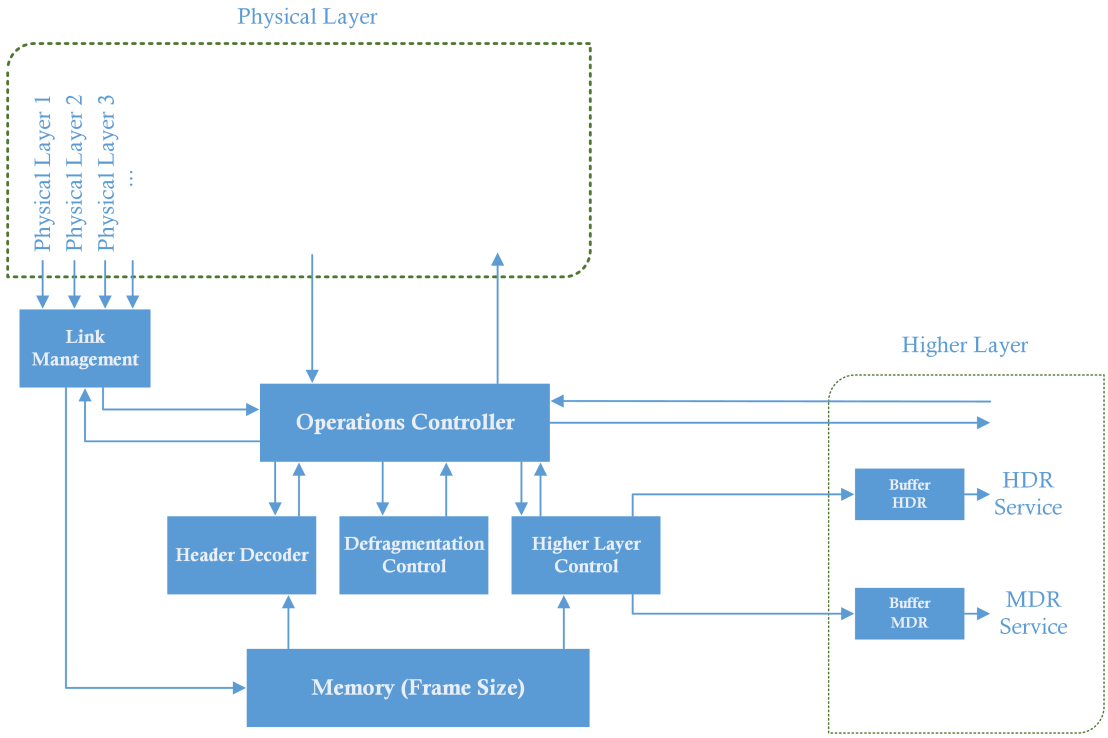

Figure 3.23: Receiver DLL

The Operations Controller is once again the "brain" block that controls the remaining blocks. Its usage in the Receiver DLL ensures the proper functioning of the receiver. The Operations Controller flowchart is represented in figure 3.24 and the reader can already observe that it is simpler than the Operations Controller presented for the Emitter DLL. As previously said, the maximum number of Higher Layer requests considered in the frame header is 32 and with this in mind, a buffer with 32 positions is created to contain all request sizes. In the asynchronous architecture of VLCLighting each block at the receiving part must have a buffer at the input of each processing block that must never be full. With this constraint in mind, the DLL must use a flag to signal the FEC block of Almost Full buffer scenarios to indicate the stop processing condition. Since in the beginning the DLL buffer has no data, the AF flag is set to false (0).

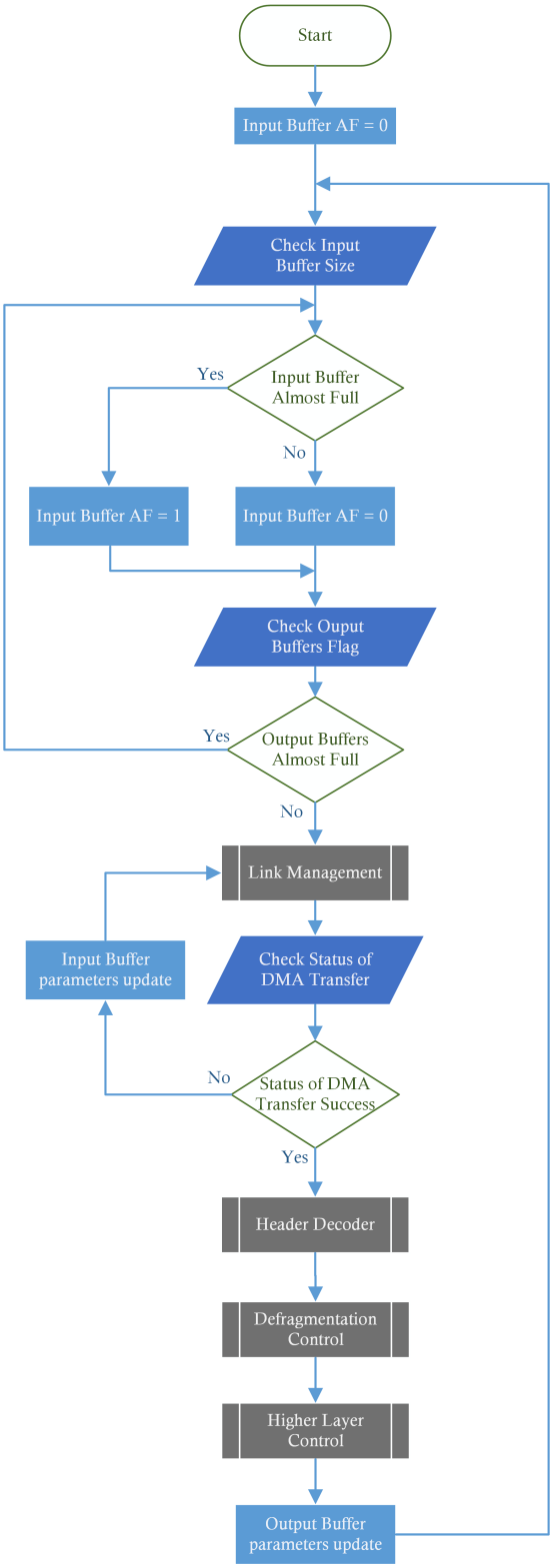

Figure 3.24: Receiver Operations Controller

The first task of the Operations Controller is to check buffer size and see if its level got to the AF state. Then, once it is sure that input buffer is capable of receiving more DLL frames from the FEC block, it checks the output buffers flag from the Higher Layer Services to see if they are capable of handling more data transfers. After this, the Link Management function is called to transfer one frame from the input buffer to the buffer intended for the temporary frame. Link Management returns with the information of data transfer completion success. If the transfer had errors, the frame is discarded and a new frame is fetched from the input buffer. After having the frame in the DLL buffer, Header Coder, Defragmentation Control and Higher Layer Control functions are called. Once the frame is transformed into Higher Layer data and inserted in the respective Services, all Buffer parameters need to be updated, this concludes one Operations Controller iteration. Now that the reader has an idea of the data flow in the receiver and Operations Controller tasks, a detailed explanation of the remaining blocks will be performed with the help of flowcharts.

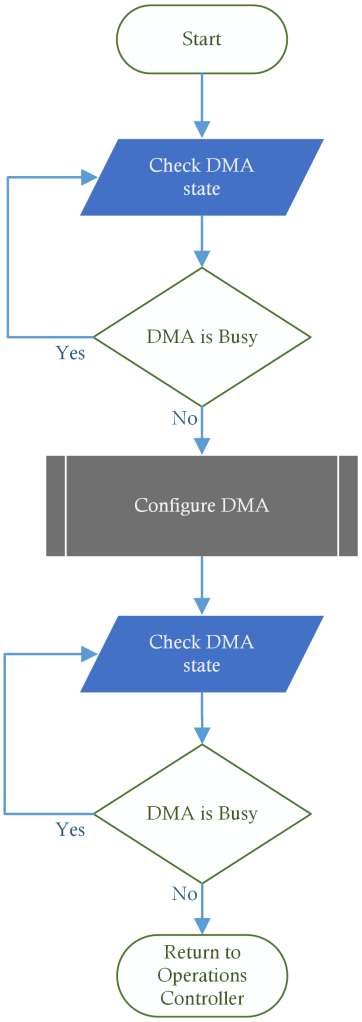

Figure 3.25: Receiver Link Management

The Link Management (seen in figure 3.25) function starts by checking if DMA is being used. If DMA is busy, the Link Management waits until it is able to configure the DMA with

the transfer parameters given by the Operations Controller. These given transfer parameters when set are responsible for the transfer between the input buffer and the DLL buffer. Once configuration is done, Link management waits and reports to Operations Controller the result.

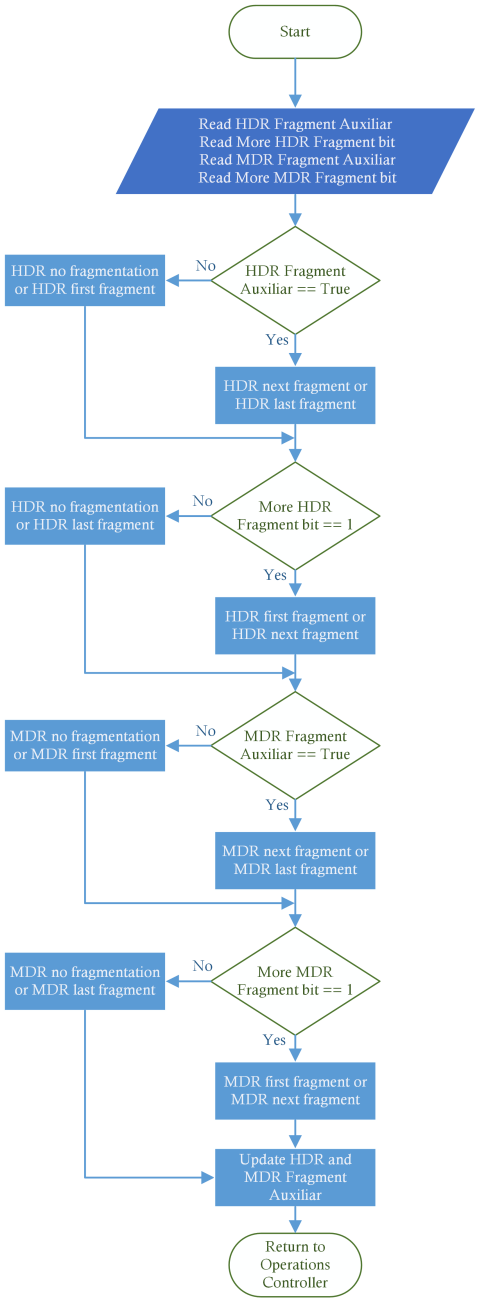

Figure 3.26: Receiver Defragmentation Controller

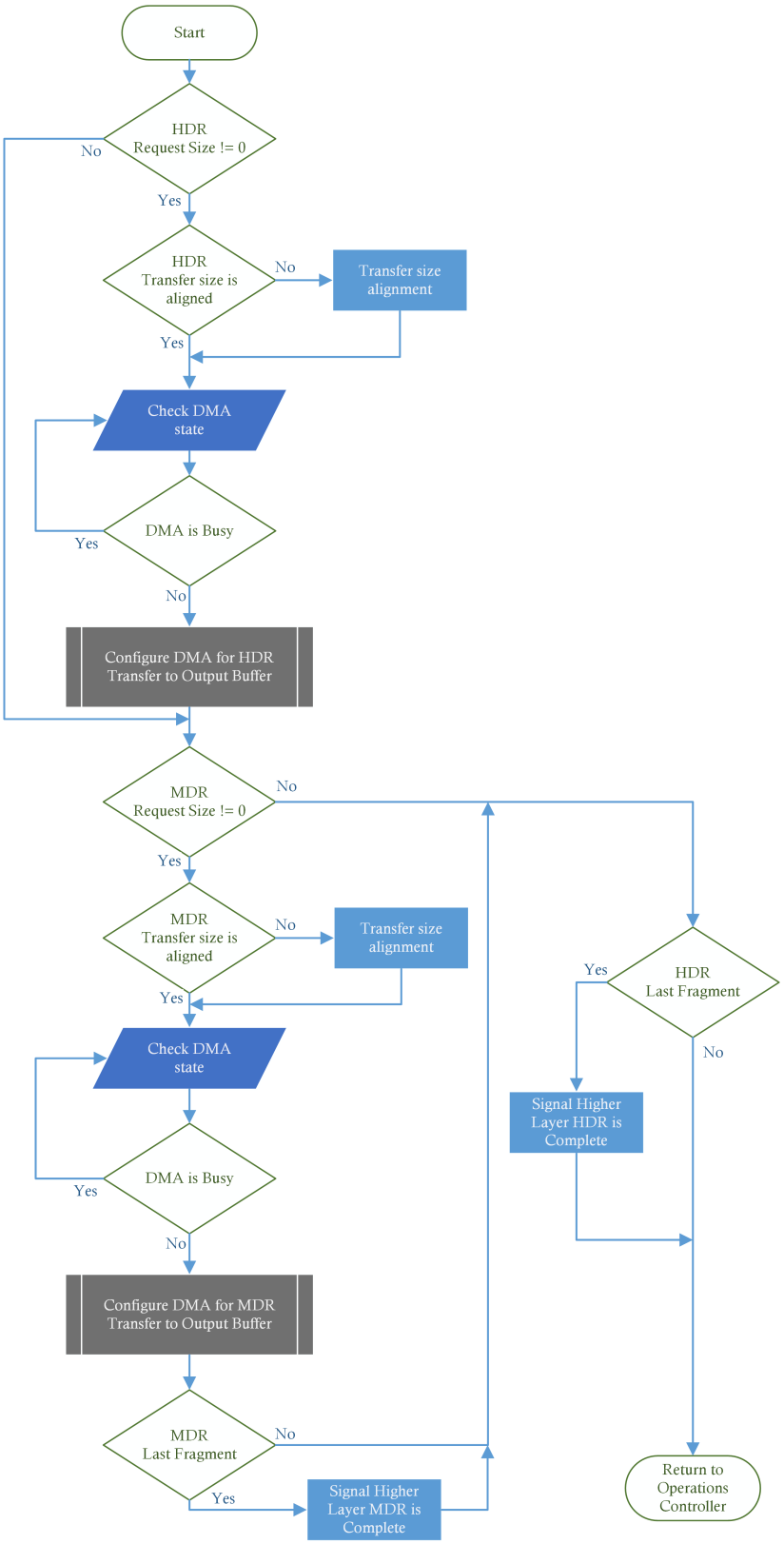

Figure 3.27: Receiver Higher Layer Controller

The next block is the Header Decoder that is responsible for header interpretation. The header interpretation will give the data size to be transferred to Higher Layer Services and provides information regarding the correct data reception. If the header contains a non-expected fragment size, the frame is incorrect and needs to be discarded.

The Defragmentation Controller block is responsible for data counting while receiving frames with fragmented requests. This means that the Defragmentation Controller block is responsible for the re-assembly parameters calculation of the fragmented Higher Layer data. Its behaviour can be seen in figure 3.26. In order to check the fragmentation information, Defragmentation Controller needs to possess auxiliary variables that store the HDR and MDR *More Fragment* parameters from the last frame. These auxiliary variables are crucial to understand which of the four possible cases of one Service Fragmentation request is in the frame (no fragmentation, first fragment arrival, last fragment arrival and next fragment arrival).

The Higher Layer Controller behaviour can be seen in figure 3.27. This block is responsible for transferring the Higher Layer data to the corresponding Service. This is performed with the parameters calculated in the Defragmentation Controller function. The first operation performed in Higher Layer Controller is to check if a Higher Layer request was in the received frame. Then, it aligns the transfer size parameter given by the Defragmentation Controller, to configure the DMA. Before configuring the DMA, the Higher Layer Controller checks if DMA is busy and if required, waits for its transfer conclusion. The first data transfer made is from the HDR Service requests and the same process occurs to the MDR Service requests. After transferring both, the Higher Layer Controller checks the calculated flag from the Defragmentation Controller. If the flag says that the fragment is the last one, it signals Higher Layer Services of fragmentation completion.

Chapter 4

DLL implementation on FPGA

In order to design the DLL that was discussed in the previous chapter, several key concepts and design tools proved to be fundamental. This chapter will provide a brief overview of FPGA basic blocks, FPGA design flow, and Xilinx ISE tools.

A FPGA is a programmable logic device based on a matrix of Configurable Logic Blocks (CLBs) that are connected through programmable interconnects. FPGAs can be programmed to the desired application or functionalities, as opposed to Application Specific Integrated Circuits (ASICs), where the device is custom built for the particular design. There are, however, One-Time Programmable (OTP) FPGAs. The majority are SRAM-based which can be reprogrammed as many times as required in the design. CLBs are the basic logic unit that consists of a configurable switch matrix with 4 or 6 inputs, some selection circuitry and flip-flops. Their number varies from device to device. While the CLB provides the logic capability, the interconnects provide the routing between CLBs and the Inputs/Outputs for the signals. The design software makes the interconnect routing task transparent to the user, thus reducing the design complexity in such devices [54].

In a microcontroller the chip is already designed to be used so, software can be written, compiled into a hex file and loaded into it without requiring hardware setup. The microcontroller stores the program in flash memory and keeps it until it is erased or replaced, meaning that with microcontrollers we only have control over the software. On the other hand, FPGAs offer higher level of configuration, because we also have control over the hardware. To create an FPGA design, Hardware Description Language (HDL) is normally used to describe the hardware needed to meet project needs. This HDL is then synthesized into a bit file that is used to configure the FPGA, providing full control over the hardware. One of the very interesting things about FPGAs is that a processor can be designed and used to run software specific tasks. In fact many companies that design digital circuits often use FPGAs to prototype their chips before creating them, reducing tremendously design costs. However FPGAs when compared to microcontrollers have a superior power consumption as well as high learning curve due to their design complexity.

The FPGA is a prime part of the proposed VLCLighting architecture due to their high flexibility and rapid prototyping capabilities. Their usage allows a reasonable cost system construction when compared to other existing real-time VLC demonstrators, which is one of the thriving reasons of VLCLighting project. It also presents high speed signal processing capabilities and high integration as well as debug tools that help their usage.

4.1 Overview of Spartan605 evaluation kit

In order to explain the DLL implementation properly, an overview of the kit that was used, will be presented. An evaluation kit from the famous brand Xilinx was used in the design of this work. Xilinx offers a comprehensive number of FPGAs to address the requirements of a wide set of applications, and these are divided in four "families" (Spartan (low-power), Artix (low-cost), Kintex (mid-range) and Virtex (high-performance)).

The evaluation kit used at the Integrated Circuits lab in the Telecommunications Institute of Aveiro was the Spartan-6 FPGA SP605, that can be seen in figure 4.1. This kit is part of the Spartan series that targets applications with a low-power footprint, extreme cost sensitivity and high-volume. The Spartan-6 family is built on a $45nm$, 9-metal layer, dual-oxide process technology. "It comes in LX and LXT versions, where the LX includes DSP slices for fast parallel ALU work and the LXT includes high-speed serial communications" [55]. Being part of Xilinx All Programmable low-end portfolio, Spartan-6 FPGAs offer advanced power management technology, up to 150K 6-input CLBs, integrated PCI Express blocks, advanced memory support, $250MHz$ DSP slices, and 3.2Gbps low-power transceivers.

The Spartan-6 FPGA SP605 Evaluation Kit enables hardware and software developers to create or evaluate designs targeting the Spartan-6 XC6SLX45T-3FGG484 FPGA, that delivers an optimal balance of low risk, low cost, and low power for cost-sensitive applications. The SP605 Evaluation Kit contains all basic components, Intellectual Property (IP) cores and software design tools to explore all its reconfigurability capability. This evaluation kit was used in the DLL, FEC and modulation with the already discussed asynchronous architecture that enables the reusage of these blocks in many other FPGA families. This SP605 kit provides a flexible environment for system design with pre-verified reference designs and examples on how to explore its features, such as high-speed serial transceivers, PCI Express, DVI, a 128MB DDR3 Component Memory, a USB JTAG Download Port, a Serial USB-UART, an Ethernet RJ45 and three oscillators, being the main one a differential 200MHz oscillator. This kit also includes an industry-standard FPGA Mezzanine Card (FMC) connector that was used to connect the ODAC to the development board.

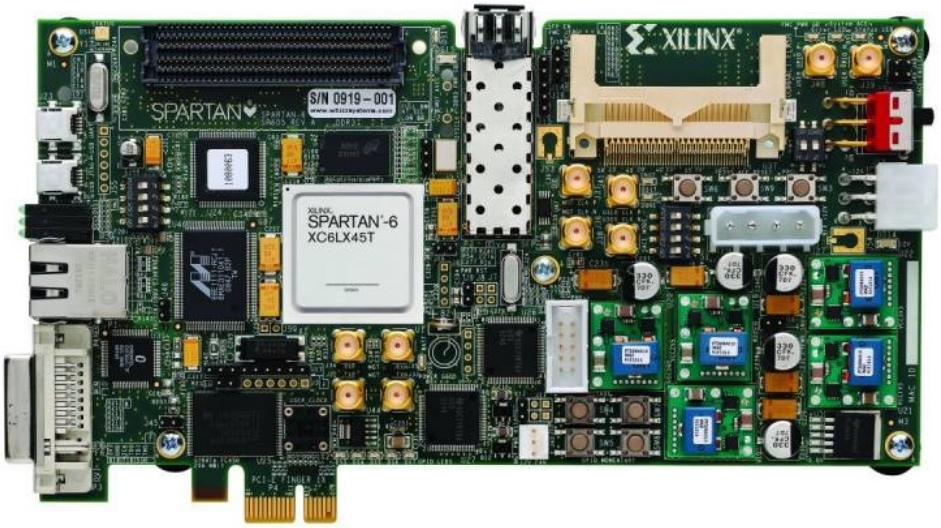

Figure 4.1: Spartan605 Evaluation Kit

Before designing the embedded processor system, some background information needs to be provided to inform the reader about the processor to be used and some items about the Xilinx EDK software tools. The microprocessors available for use in Xilinx FPGAs with Xilinx EDK software tools can be broken down into two broad categories. There are soft-core microprocessors (MicroBlaze) and the hard-core embedded microprocessor (PowerPC). This study will only focus on the soft-core MicroBlaze microprocessor, which will be discussed in the next section.

4.1.1 Microblaze architecture

The DLL implementation was accomplished in an embedded design with the Microblaze soft processor core. It was imperative that some of Microblaze characteristics were explored before hands. The MicroBlaze embedded processor soft core is a Reduced Instruction Set Computer (RISC) optimized for implementation in Xilinx FPGAs. Its functional block diagram can be seen in figure 4.2.

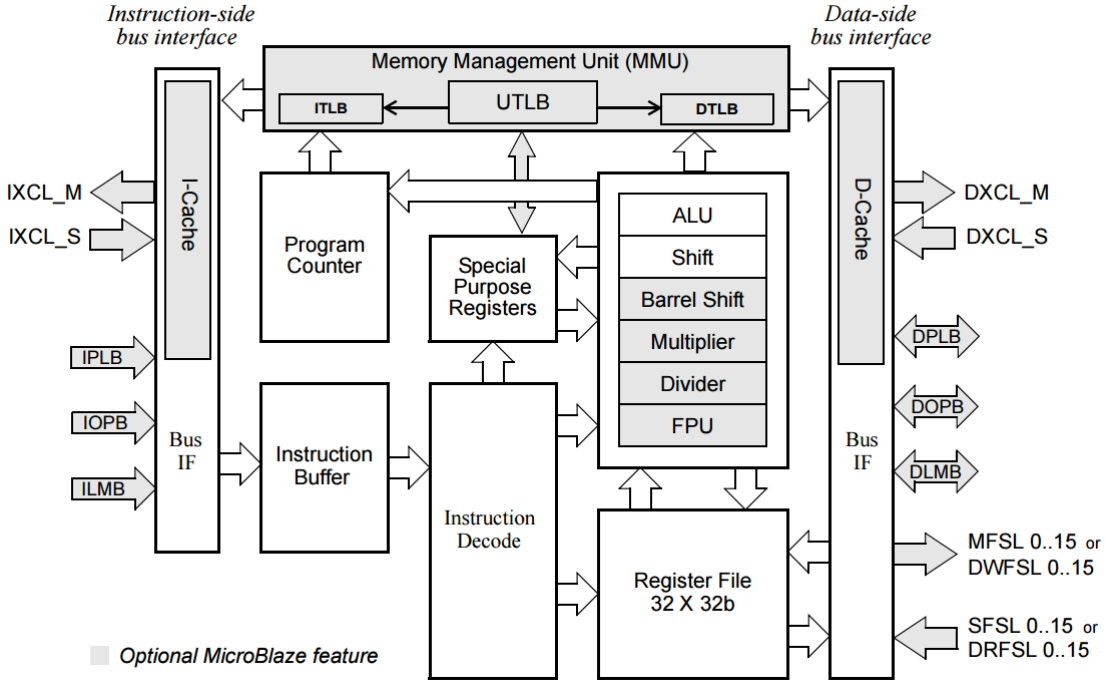

Figure 4.2: Microblaze architecture [56]

The MicroBlaze is a virtual microprocessor that is built by combining blocks of code called cores inside a Xilinx FPGA. This allows the developer to tailor the project according to his needs. Microblaze is an FPGA-based soft processor with advanced architecture options like Advanced eXtensible Interface (AXI) and Programmable Logic Block (PLB) interfaces, Memory Management Unit (MMU), instruction and data-side cache, configurable pipeline depth, Floating-Point unit (FPU), and much more. Microblaze is a 32-bit RISC Harvard architecture soft processor core that is included free in Xilinx Design Software (Vivado or the older Xilinx EDK). It is a highly flexible architecture with a rich instruction set optimized for embedded applications, that can deliver the exact processing system required at the lowest

system cost possible.

As it can be seen in figure 4.2, the Microblaze architecture possesses an Instruction bus interface and a separate Data bus-interface connected to the MMU. An instruction is fetched from the MMU and then processed according to its specifications.

It is important to note that Microblaze has an orthogonal instruction set architecture with thirty-two 32-bit general purpose registers and up to eighteen 32-bit special purpose registers, but their numbers depend on configured options. Later, in this study Program Counter (PC) will be mentioned, so it is of particular interest that the reader knows that it is part of the Special Purpose Registers that holds the memory 32-bit address of the current instruction being executed. All MicroBlaze instructions are 32 bits and are defined as either Type A or Type B. Type A instructions have up to two source register operands and one destination register operand. Type B instructions have one source register, a 16-bit immediate operand (that can be extended to 32 bits) and a single destination register operand. Instructions can be categorized in arithmetic, logical, branch, load/store, and special. MicroBlaze instruction execution is pipelined. For most instructions, each stage takes one clock cycle to complete. Consequently, the number of clock cycles necessary for a specific instruction to complete is equal to the number of pipeline stages, one instruction is completed on every cycle. Few instructions require multiple clock cycles in the execute stage to complete, this is achieved by stalling the pipeline. When executing from a slower memory, instruction fetches may take several cycles which introduce additional latency and affect the efficiency of the pipeline. In order to overcome this issue, MicroBlaze implements an instruction prefetch buffer that reduces the impact of such multi-cycle instruction memory latency [57].

MicroBlaze is implemented with a Harvard memory architecture, which means that instruction and data accesses are done in separate address spaces. Each address space has a 32-bit range, that is able to handle up to 4GB of instructions and data memory respectively. The instruction and data memory ranges can be made to overlap by mapping them both to the same physical memory, a feature useful for software debugging. As previously said, a major concern to take into account is address alignment. Data accesses must be aligned (word accesses must be on word boundaries, halfword on halfword boundaries), unless the processor is configured to support unaligned exceptions (which decreases tremendously its performance). So, all instruction accesses must be word aligned.

In this Microblaze architecture, data accesses are not separated between Inputs/Outputs and memory. The processor has up to three interfaces for memory accesses: Local Memory Bus (LMB), AXI for peripheral access and AXI or AXI Coherency Extension (ACE) for cache access [57].

4.1.2 Advanced eXtensible Interface

Advanced eXtensible Interface 4 (AXI4) is the fourth generation of the AMBA (a family of micro controller buses) interface specification from ARM. Xilinx offers a broad set of AXI4 based IPs with a single open standard interface across the Embedded, DSP, and Logic domains.

AXI4 provides improvements and enhancements across the board, providing benefits to Productivity, Flexibility and Availability. Due to the current standardizing on the AXI interface, learning only a single protocol for IP usage is required, increasing productivity. The AXI4 provides flexibility for the required application with its three subsets (AXI4, AXI4-Lite, AXI4-Stream). The understanding of these subsets and their application scenarios was cru-

cial for this work. "The AXI4 is for memory mapped interfaces and allows burst of up to 256 data transfer cycles with just a single address phase. AXI4-Lite is a light-weight, single transaction memory mapped interface. It has a small logic footprint and is a simple interface to work with both in design and usage. AXI4-Stream removes the requirement for an address phase altogether and allows unlimited data burst size. AXI4-Stream interfaces and transfers do not have address phases and are therefore not considered to be memory-mapped" [58]. Its usage benefits with availability because by moving to an industry-standard, developers can have access to an worldwide community of ARM Partners and not being restricted to Xilinx IP catalogue .

Its adoption makes all interface subsets use the same transfer protocol while including standard models and checkers for designers usage. It provides a unified interface on IP across communications, video, embedded and DSP functions and enables Xilinx to efficiently deliver enhanced native memory, external memory interface and memory controller solutions across all application domains.

"In memory mapped AXI, all transactions involve the concept of a target address within a system memory space and data to be transferred. Memory mapped systems often provide a more homogeneous way to view the system, because the IPs operate around a defined memory map" [58]. AXI specifications describe an interface between a single AXI master and a single AXI slave, that represent different IP cores exchange of information. The connection between AXI masters and slaves is performed using an Interconnect. "The Xilinx AXI Interconnect IP contains AXI-compliant master and slave interfaces, and can be used to route transactions between one or more AXI masters and slaves" [58]. It is very important to remark that the data can move between the master and slave in both directions simultaneously, and that their transfer sizes can vary. As previously said, the AXI4 has a burst transaction limit of 256 data transfers while AXI4-Lite interface only allows 1 data transfer per transaction. The AXI4-Stream protocol defines a single channel for transmission of streaming data and can burst an unlimited amount of data. Bidirectional data transfer can be achieved with AXI4 usage, because it provides separate data and address connections for reads and writes. AXI4 can also specify different clock for each AXI master-slave pair at hardware level.

The AXI was used together with Xilinx IP cores (will be specified later on) to develop the DLL on the SP605.

4.1.3 Memories

SP605 Evaluation Board contains four memories that are the 128 MB DDR3 component memory, the 32 MB parallel (BPI) flash memory, the 8 Kb IIC EEPROM and the 8 MB Quad SPI flash memory. It also comes with an Xilinx System ACE CompactFlash (CF) that allows a CompactFlash card to be used to expand board memory. As the reader will see, DDR3 was the main memory block used in this work. As the User Guide specifies, "the 1-Gb Micron MT41J64M16LA-187E (96-ball) DDR3 memory component is accessible through Bank 3 of the LX45T device, where the Spartan-6 FPGA hard memory controller is used for data transfer across the DDR3 memory interfaces 16-bit data path using SSTL15 signaling" [59].

The Microblaze processor has up to three interfaces for memory accesses that are called Local Memory Bus (LMB), Processor Local Bus (PLB) or On-Chip Peripheral Bus (OPB) and Xilinx CacheLink (XCL). The LMB memory address range must not overlap with the range of the remaining. The LMB is a synchronous bus primarily designed to have fast

access, typically to on-chip block RAM (BRAM) memories. The PLB and OPB represent general purpose bus interfaces to on-chip or off-chip memories, as well as other non-memory peripherals. The Xilinx CacheLink interface is a high performance solution intended for specialized external memory controllers.

MicroBlaze has a single cycle latency for accesses to local memory (LMB) and for cache read hits, except with area optimization enabled when data side accesses and data cache read hits require two clock cycles. A data cache write normally has two cycles of latency. The MicroBlaze instruction and data caches can be configured to use 4 or 8 word cache lines. When using a longer cache line, more bytes are prefetched, which generally improves performance for software with sequential access patterns. However, for software with a more random access pattern, the performance can be decreased for a given cache size. This is caused by a reduced cache hit rate due to fewer available cache lines [56].

Local memory is used for data and program storage and is implemented using Block RAM. The size of the local memory is parameterized between 4kB and 64kB. The local memory is connected to MicroBlaze through the Local Memory Bus (LMB) and the LMB BRAM Interface Controllers [60].

MicroBlaze can be used with an optional data cache for improved performance. It is important that the cached memory range must not include addresses in the LMB address range. MicroBlaze may be used with an optional instruction cache for improved performance when executing code that resides outside the LMB address range. When the instruction cache is used, the memory addresses space is split into two segments: a cacheable segment and a non-cacheable segment. The cacheable segment is determined by two parameters: C_ICACHE_BASEADDR and C_ICACHE_HIGHADDR. All addresses within these ranges correspond to the cacheable address segment. All other addresses are non-cacheable. The MicroBlaze instruction cache can be configured from 2kB to 64 kB. For every instruction fetched, the instruction cache detects if the instruction address belongs to the cacheable segment. If the address is non-cacheable, the cache controller ignores the instruction and lets the OPB or LMB complete the request. If the address is cacheable, a lookup is performed on the tag memory to check if the requested address is currently cached [56].

MicroBlaze can be configured to cache data over either the OPB interface, or the dedicated Xilinx CacheLink interface. CacheLink uses dedicated interface for memory accesses, thus reducing the traffic on the OPB.

4.1.4 Xilinx ISE Design Suite

Xilinx ISE Design Suite is the software that supports Spartan 6 FPGA family. It is important to regard that in order for this Design Suite to work properly, a set of software requirements need to be fulfilled. Therefore, the Xilinx ISE Design Suite used in this work, was installed in a Windows 7 machine.

The ISE Design Suite (Embedded Edition) is the development package for building MicroBlaze embedded processor systems in Xilinx FPGAs. It includes XPS, SDK, a large repository of plug and play IP that include MicroBlaze Soft Processor and peripherals, and a complete Register-Transfer Level (RTL) to bit stream design flow. This Design Suite provides the fundamental tools, technologies and familiar design flow to achieve optimal design results. These include intelligent clock gating for dynamic power reduction, team design for multi-site design teams, design preservation for timing repeatability and a partial reconfiguration option for greater system flexibility, size, power and cost reduction [61].

XPS is used to configure and build the hardware specification of their embedded system (processor core, memory-controller, I/O peripherals, etc.). The XPS converts the designer's platform specification into a synthesizeable RTL description, and writes a set of scripts to automate the implementation of the embedded system (from RTL to a bitstream file). RTL abstraction is used in hardware description languages HDLs, like Verilog and VHDL to create the high-level representations of a circuit (RTL). For the MicroBlaze core, the EDK normally generates an encrypted netlist, but the processor description (written in VHDL) can be purchased from Xilinx. It is also worth mentioning that XPS provides a block-based system assembly tool for connecting blocks of IPs together using many bus interfaces (including AXI) and create embedded systems. In other words, XPS provides a graphical interface for connection of processors, peripherals, and bus interfaces [58, 61].

On the other hand, the SDK is responsible for creating embedded applications for the designed embedded system. It is built on Eclipse 4.3.2 and it directly interfaces with the XPS embedded hardware design. SDK offers editors, compilers, build tools, flash memory management, full suite of libraries and device driver, and JTAG/GDB debug integration. Powered by the GNU toolchain (GNU Compiler Collection, GNU Debugger), the SDK enables programmers to write, compile, and debug C/C++ applications for their embedded system. Xilinx includes a cycle-accurate instruction set simulator (ISS), giving programmers the choice of testing their software in simulation or using a suitable FPGA-board to download and execute on the actual system. It is also noteworthy that SDK is the software development environment for application projects. The integration between XPS and SDK in AXI-based embedded systems is achieved with XML format export of the hardware platform specifications [58, 61].

"Xilinx IP cores and host drivers supply a simple end-to-end solution for the data transport. Together with High-Level Synthesis (HLS), a fullblown coprocessing system having a simple programming interface can be set up without any FPGA expertise" [62]. Xilinx and its partners offer hundreds of free and for-purchase IP, verified and guaranteed to meet timing parameters, thus speeding up the design cycle and allowing you to focus on the value add components of the design instead of standards compliance [54].

4.2 Embedded design process

The tools provided in EDK are designed to assist in all phases of the embedded design process, as illustrated in Figure 4.3 [63]. Before going to the process flow detail, an overview on the typically generated design files, as well as, their purpose, is mandatory.

An embedded hardware platform typically consists of one or more processors, peripherals and memory blocks, interconnected via processor buses. Each of the processor cores (also referred to as pcores or processor IPs) has a number of parameters that can be adjusted to customize its behaviour, which can also define the address map of peripherals and memories. XPS lets the designer select from various optional features. This allows an implementation of a subset of the FPGA functionalities required by the desired application [64]. These bus architecture, peripherals, processor, system connectivity and address space characterization are made in the graphical interface of XPS, so that then a Microprocessor Hardware Specification (MHS) file is generated. The MHS serves as an input to the Platform Generator (Platgen) tool. On the other hand, the port listing and default connectivity for bus interfaces are defined in the Microprocessor Pheripheral Definition (MPD) file. It is very important to

regard that any MPD parameter is overwritten by the equivalent MHS assignment. Another important file configuration is the User Constraints File (UCF) that specifies timing and placement constraints for the FPGA design [65]. All the above mentioned files constitute the input files of XPS Project.

The Xilinx Microprocessor Project (XMP) file is the top-level project file for an EDK design and is used for project management. The Base System Builder (BSB) output file together with XMP constitutes the general project files [65].

Some of the XPS output files that are worth mentioning are the BMM, the ELF, the MHS, the NGC and XML. A Block RAM Memory Map (BMM) file contains a syntactic description of how individual block RAMs constitute a contiguous logical data space. The Executable and Linkable Format (ELF) is a common standard in computing. An executable or executable file, in computer science, is a file whose contents are meant to be interpreted as a program by a computer. Most often they contain the binary representation of machine instructions of a specific processor, but can also contain an intermediate form that requires the services of an interpreter to run. The MHS file defines the hardware component and serves as an input to the Platform Generator (Platgen) tool. The NGC file is a netlist that contains logical design data and constraints. This file replaces both Electronic Data Interchange Format (EDIF) and Netlist Constraints File (NCF) files [65]. The XML file contains all of the hardware specification necessary to the software development and deploy software applications for the FPGA.

The process flow steps can be followed in figure 4.3.

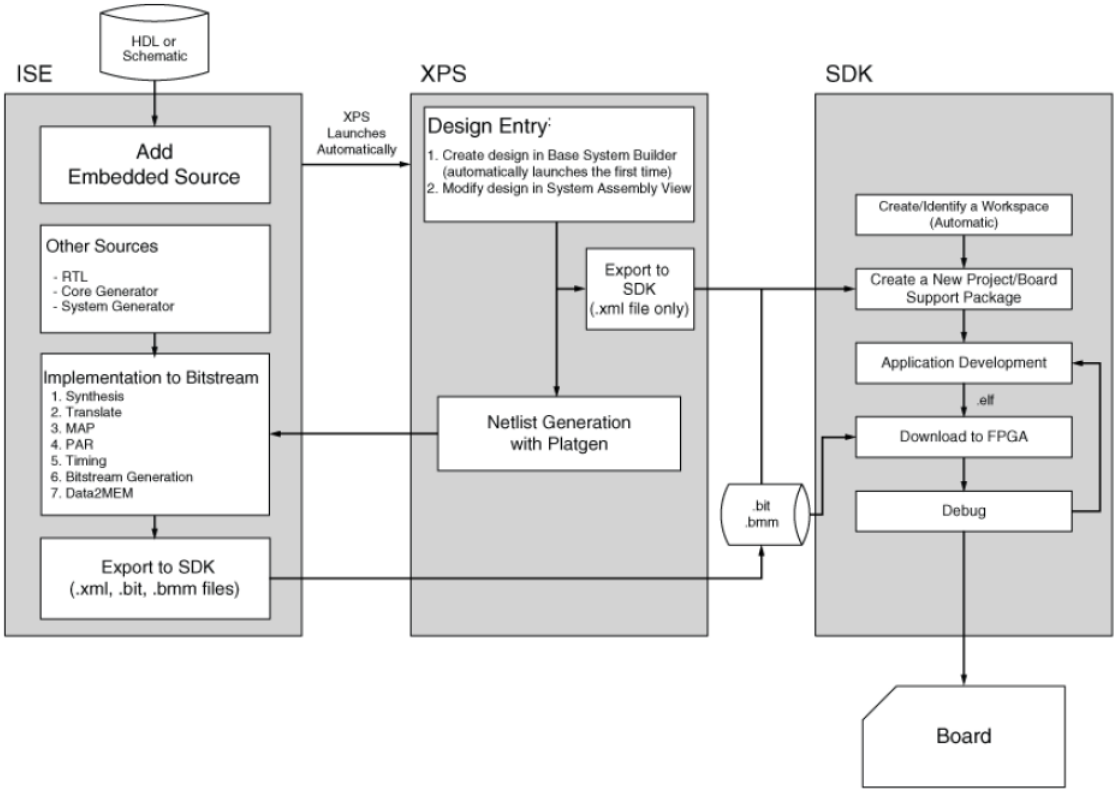

Figure 4.3: Basic Embedded Design Process Flow [63]

Typically, the ISE development software is used to add an Embedded Processor source, which is then created in XPS using the Base System Builder. Then hardware is inserted from the Xilinx provided IP Catalogue or if extra customization is required by the developer. A Custom IP can be created with Create and Import Peripheral Wizard of XPS. Once an IP Core is added to the system, it is included into the MHS file. After adding an hardware to the system, the next step is to make sure that it is connected through a bus instance to the remaining ones, according to the system design aim. Next, address assignment is required, where specific address ranges are defined for the communication between microblaze and external devices, performed through the use of registers or memories. Now that instances are connected through buses with their addresses assigned, configuration of the inserted IPs can be performed to meet the desired system requisites. After this, ports parameters are assigned in the Net field of XPS, where external ports in the design need to be set as external. As previously said, the UCF specifies timing and placement constraints of the design, and its editing is the next step in the design (Pin constraints present in the Board User Manual are here entered). With MHS and MPD files set, hardware netlists are generated with PlatGen. PlatGen customizes and generates the embedded processor system through the use of hardware netlists HDL files. PlatGen also generates the system-level HDL file that interconnects all the IP cores, which can then be synthesized as part of the overall design flow. After the hardware platform design is complete. The FPGA configuration bitstream (.bit) can be generated with the Generate Bitstream command in XPS. However, this hardware bitstream is not ready to be applied to an FPGA until a software component of the embedded system is made. The Export to SDK command exports the Hardware Platform Description (XML - system.xml) to the SDK, so that the development of the desired embedded software application can be performed, and the bistream and BMM files downloaded to the FPGA.

The SDK application development starts with the identification of the hardware configuration from the Hardware Platform Specification (XML). Before being able to create software applications in SDK, a Board Support Package (BSP) needs to be created. BSP is the collective term referring to all of the software components required to match a given operating system to a given hardware design. It is a collection of libraries and drivers that will form the lowest layer of the developed application software stack.

There are two board support packages available for application development (Standalone and Xilkernel). Standalone is a single-threaded, simple Operating System platform that provides the lowest layer of software modules used to access processor-specific functions (some typical functions include setting up the interrupts and exceptions, configuring caches, and other hardware specific functions) [66]. On the other hand, Xilkernel is a simple embedded processor kernel that can be customized to a large degree for a given system (some of its key features are multi-tasking, priority-driven pre-emptive scheduling and inter-process communication) [66]. Due to their different main goals, the natural choice for the project was the Standalone, because not many processes will be running in the DLL at a given time. The Xilkernel choice would not be beneficial, because its advantages (e.g. multi-taking) would only bring higher complexity with no real extra value for the proposed work.

The next step in the application development is to create a software project attached to this BSP project, thus concluding the SDK setup for the application programming. After compiling the C/C++ application code with a platform-specific gcc/g++ compiler (Figure 4.4), the object files from the application are linked to the BSP, resulting in an ELF file creation. This step is performed by a linker which takes as input a set of object files and a linker script that specifies where object files should be placed in memory [67]. SDK provides

a linker script generator, where the only action required is to assign different code and data sections in the ELF file to different memory regions.

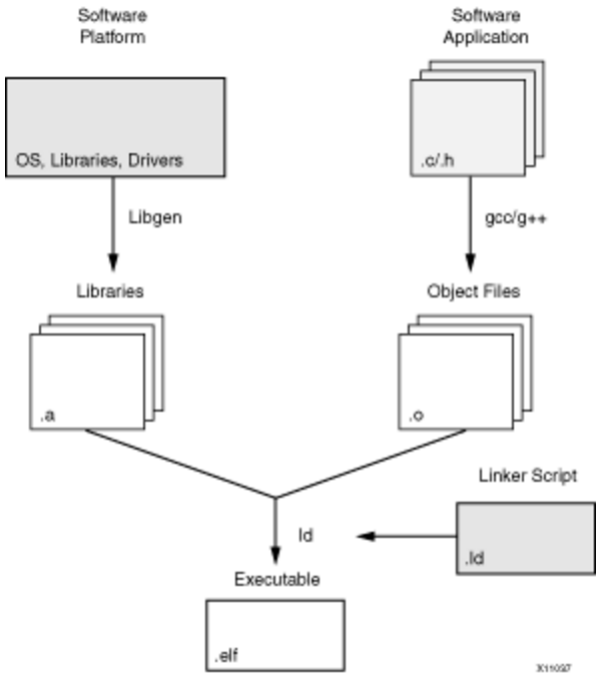

Figure 4.4: Software workflow in SDK [67]

Once the FPGA is configured with the bitstream containing the embedded design, the ELF file from the software project can be downloaded and debugged. It is important to note that for a given application, a Release and Debug configuration can be made, being that the Debug is created by default. Debugging is the process of downloading and running C/C++ code on the target hardware to evaluate whether the code is performing correctly. Because this is an FPGA, configuring the FPGA with a bitstream that loads a design into the FPGA is required even when debugging. The process of programming FPGA is achieved with the USB JTAG Download Port and its control is accomplished with the USB UART Cable between the demo board and the computer [63].

4.3 Xilinx Platform Studio

XPS provides a set of IP cores that were used to expedite the embedded design process flow and accomplish the proposed work. Since this DLL will follow the structure already mentioned in section 3.3, where the main goal is data transfer between different memory addresses or even different memories, the first step was to understand all Xilinx memory and respective IP Controllers available for the development board and identify which ones comply with the structural requirements.

One of the essential devices for maximizing performance in FPGA designs is the DMA Engine. DMA stands for Direct Memory Access, and a DMA engine allows the transfer of data from one part of the system to another. The simplest usage of a DMA is the transfer of data from one part of the memory to another, but a DMA engine can also be used to

transfer data from any data producer (eg. an ADC) to a memory, and even from a memory to any data consumer (eg. a DAC). This means that a DMA engine allows certain hardware to access system memory independently of the central processing unit (CPU). In interruption cases the DMA engine is configured and initialized by the CPU and while DMA engine ensures the transfer, CPU can resume to its other operations and is interrupted when operation is completed. This is particularly useful when CPU cannot keep up with the rate of data transfer, or when working with relatively slow I/O data transfers that could lead to an processor overhead increase.

Xilinx provides the developer with three different DMA engines IPs that are the AXI DMA, the AXI Video DMA and the AXI Central DMA. Each of these soft IPs use the same core (DataMover), but target to different peripherals that use different AXI protocols. The AXI DMA IP provides high-bandwidth direct memory access between memory and AXI4-Stream-type target peripherals. On the other hand, the AXI VDMA core is a soft Xilinx IP core that provides high-bandwidth direct memory access between memory and AXI4-Stream type video target peripherals. Finally, the AXI CDMA provides high-bandwidth direct memory access between a memory-mapped source address and a memory-mapped destination address using the AXI4 protocol.

Due to the asynchronous architecture (Figure 3.2) present in VLCLighting project, where each block contains a memory that is shared with the remaining blocks, the DMA choice has fallen on the AXI CDMA. The usage of the AXI CDMA will provide data transfer between DLL and its adjacent blocks, while offloading the CPU from the data transfer and enabling his usage to the control and sequencing tasks of the DLL only. While data transfer is achieved with AXI4 protocol, the initialization, status, and management registers are accessed through the use of an AXI4-Lite protocol, suitable for the MicroBlaze microprocessor. AXI CDMA supports primary AXI Memory Map data width of 32, 64, 128, 256, 512, and 1,024 bits, and also supports a Scatter Gather (SG) optional feature that increases the offloading CPU management tasks to hardware automation. As previously said, the CDMA uses the DataMover. DataMover is used to achieve high-throughput transfer of data with 4KB address boundary protection, automatic burst partitioning and to queue multiple transfer requests. One feature that was not used but that is also present in the AXI CDMA is the Data Realignment Engine (DRE), responsible for data alignment at the byte level. This CDMA IP can also perform an optional asynchronous operation mode, that signals with an interruption the Microblaze once the transfer is completed.

As stated in 3.4, a DLL frame needs to be sent within a certain time. To provide DLL with this requirement, an AXI Timer needs to be inserted in the embedded design. An AXI Timer is a 32 bit timer module that interfaces to the AXI4-Lite interface, and that can be configured with different widths. Each timer module has an associated load register that is used to hold either the initial value for the counter for event generation or a capture value, depending on the mode of the timer. The generate value, which is the value loaded into the load register, is used to generate a single interrupt at the expiration of an interval or a continuous series of interrupts with a programmable interval.

In order to use interruptions, an Interrupt Controller that concentrates the interrupt inputs from the Timer to a single interrupt output, is required. The used AXI Interrupt Controller is based on AXI4-Lite specification and it prioritizes the different interrupt requests from the different interrupt sources to then signal the Microblaze of their occurrence.

The first step when implementing the embedded design in the XPS is to set the interconnect type (AXI or PLB). The previously described AXI was the chosen one, because of the

stated reasons in section 4.1.2 that makes it the newly standard interface adopted by Xilinx and made the PLB a legacy bus standard that is not supported by newer FPGA families. After interconnect type selection, XPS needs to be set with the desired target board and System Configuration. As previously said the target board in this study is the Spartan-6 SP605 Evaluation Platform, and Single Microblaze Edition and Area Optimization Strategy constitute the System Configuration parameters. Next step is Processor, Cache and Peripheral configuration, where processor frequency was set to its maximum (100MHz), Local Memory Size set to 32KB, Instruction Cache set to 16KB, Data Cache set to 16KB and all the peripherals removed with the exception of RS232_Uart (that will be required for the board communication), and DDR3. The Local Memory Size was set to 32KB, because of a pre-warning from XPS literature concerning timing closure problems of large-sized BRAM. All these steps led to XPS Project and Design files generation, that form the base system project.

The required AXI CDMA IP and AXI Timer IP were added, from the "DMA and Timer" branch of the provided IP catalog, and the AXI Timer was added from the Clock, Reset and Interrupt branch. Once these IPs were added, their configuration was performed according to project needs.

The AXI CDMA was configured as in figure 4.5, where it is important to notice the data width of 32bits with a maximum burst of 64 transfers. The maximum burst parameter was due to the DLL frame size (208bytes), that in order to be transferred with a data width of 32bits requires a total of 52 burst transfers. It is important to regard that these values will be changed in chapter 5 to demonstrate their impact on the system performance. Include Data Realignment was left unchecked because this feature restricts the allowed data width values to 32 and 64 and increases the resources required in the system design. Scatter Gather mode is a mechanism that allows automated data transfer scheduling through a pre-programmed instruction list of transfer descriptors. AXI CDMA can perform these transfers automatically without the need to have each transfer programmed by the CPU. Since this system design constitutes a first approach and simplicity was required, Scatter Gather mode was considered to be a value added feature to be studied in future work, and was left unchecked. As recommended in XPS litterature, Store and Forward functionality was left checked because it provides data buffering and management of the DataMovers address request pipelining.

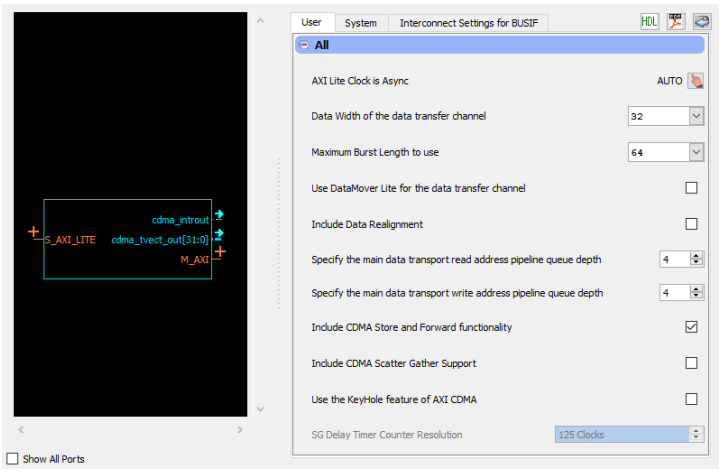

Figure 4.5: AXI CDMA Configuration

The AXI Timer configuration can be seen in figure 4.6, where most of its parameters were left as default. Since this AXI Timer was the only timer in the design, the Only Timer parameter, that can be seen in that figure, was checked.

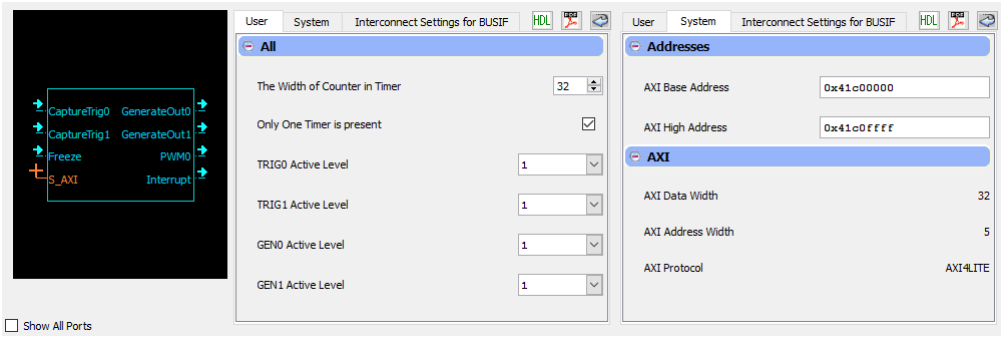

Figure 4.6: AXI Timer Configuration

The AXI Interrupt Controller setting can be seen in figure 4.7. In order to be able to read the register that indicates the presence of an active interrupt, the Interrupt Pending Register (IPR) parameter was left checked. On the other hand, Set Interrupt Enables (SIE) enables the IER (Interrupt Enable Register) bits set in a single atomic operation, rather than using a read, modify or write sequences. The reason to leave Clear Interrupt Enables (CIE) checked was similar to last one, but this operation is responsible for clearing rather than enabling the IER. Interrupt Vector Register (IVR) was left enabled to contain the ordinal value of the highest priority, enabled, and active interrupt input. The IRQ Output Use Level parameter can be edge-sensitive or level-sensitive. The difference between both is that level-sensitive stores the interrupt condition and retains it until acknowledgement even if the input level becomes inactive during this time, therefore it was the chosen one to be used.

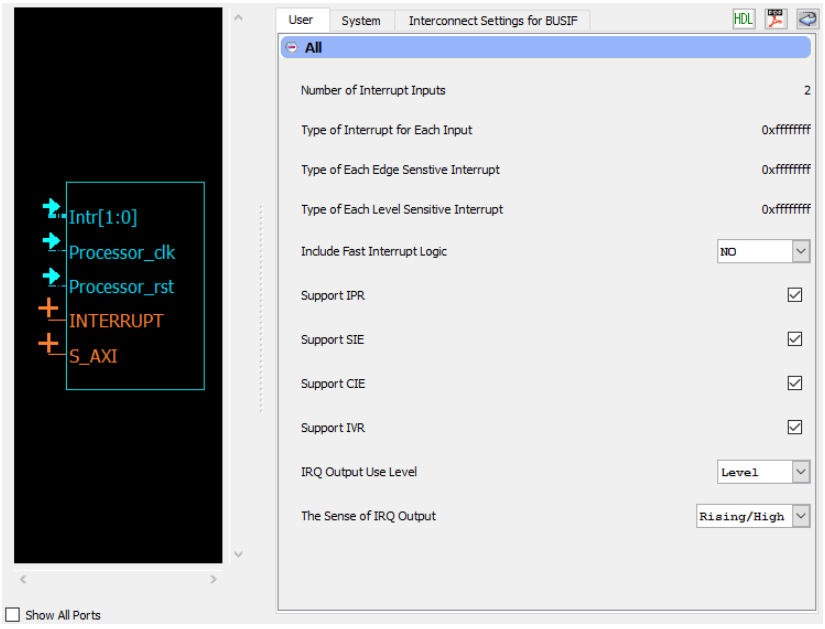

Figure 4.7: AXI Interrupt Controller Configuration

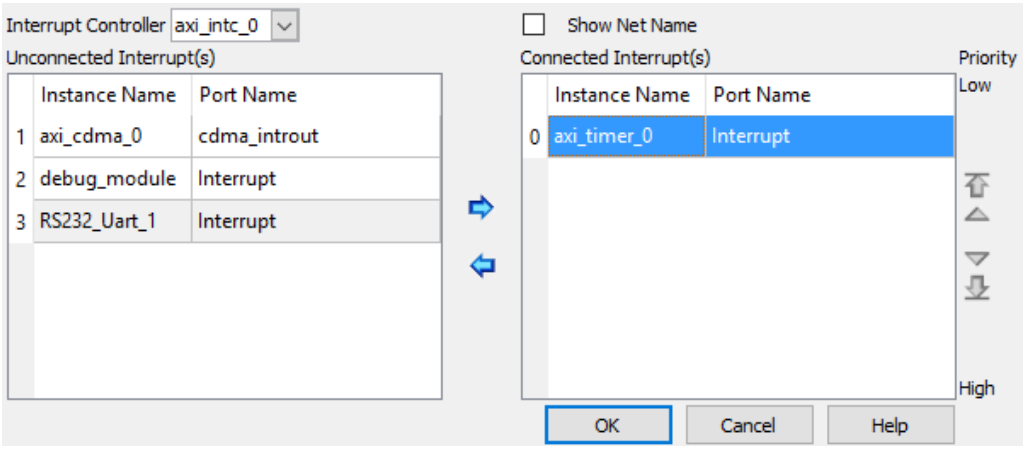

Figure 4.8: XPS Interrupt Connection Dialog

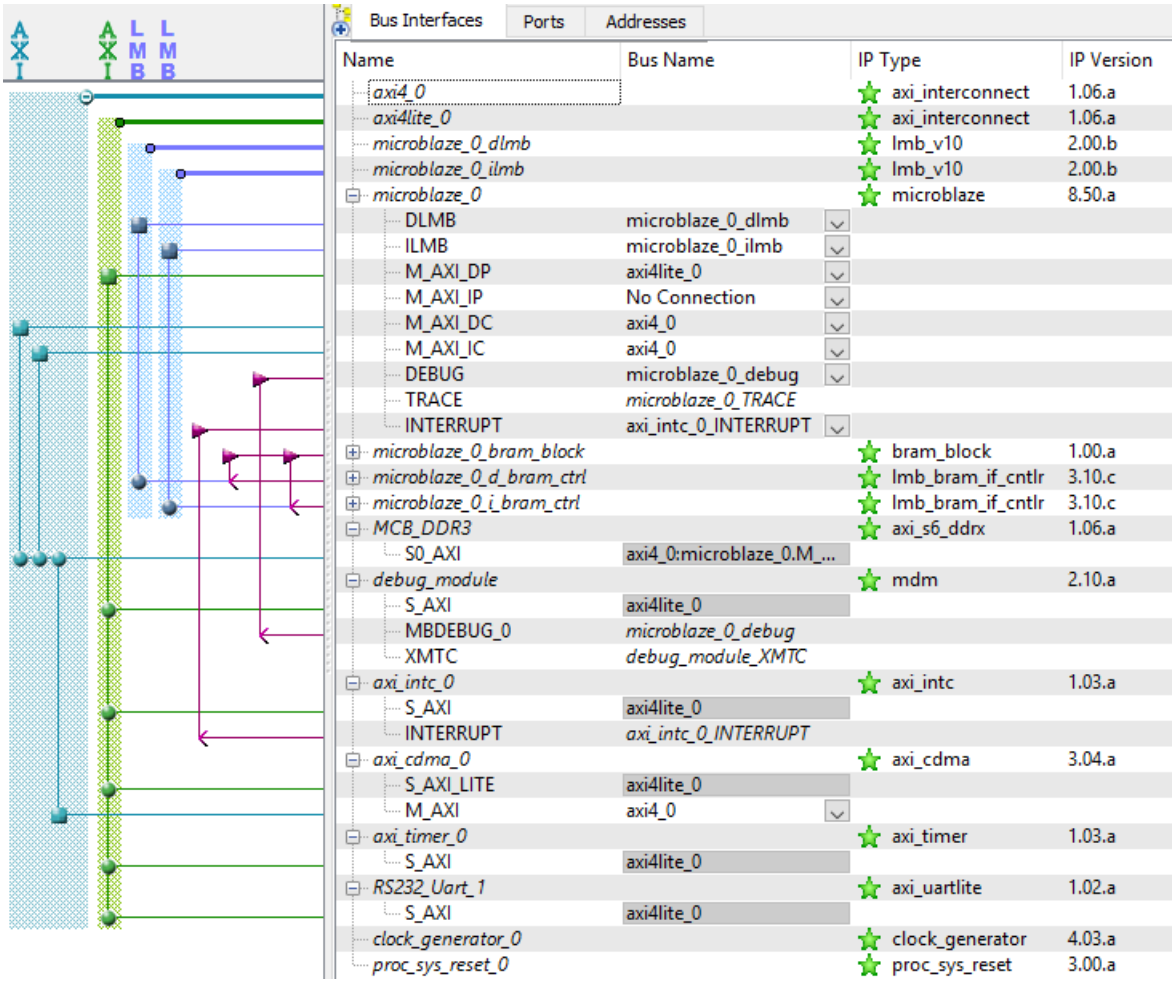

Figure 4.9: XPS System Assembly View

Now that the AXI Interrupt Controller is present in the design AXI Timer interruption needs to be connected to it. Since the DLL design requires to send a junk frame if a certain time as passed, AXI Timer needs to use an interruption (Figure 4.8). The Bus connections that ensures communication between the peripheral devices and microblaze was performed as in Figure 4.9. The microblaze interrupt port was connected to the Interrupt Controller output. Timer S_AXI was connected to Interrupt Controller S_AXI through AXI4-Lite Bus. Microblaze M_AXI_IC and M_AXI_DC were connected to DDR3 S_AXI through AXI4 Bus. The result of these connections can also be seen in XPS Ports section (Figure 4.10) that was left as default.

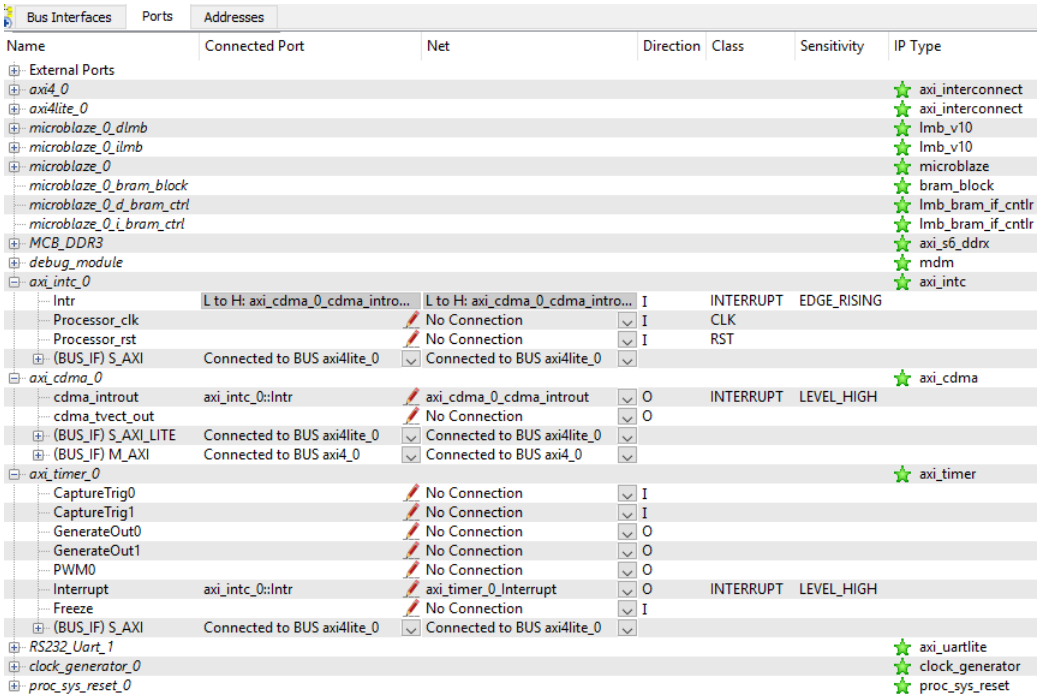

Figure 4.10: XPS Ports

The next step in the design is to assign addresses. Addresses can be inserted manually by changing the base address and then XPS will calculate the high address from base address and size entries. However, XPS presents the developer with a Generate Address button that can automate this process and automatically fill all addresses fields. The addresses that were assigned in the project can be seen in figure 4.11.

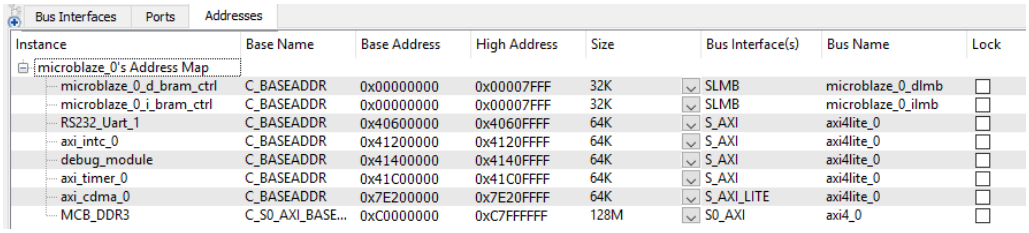

Figure 4.11: XPS Addresses

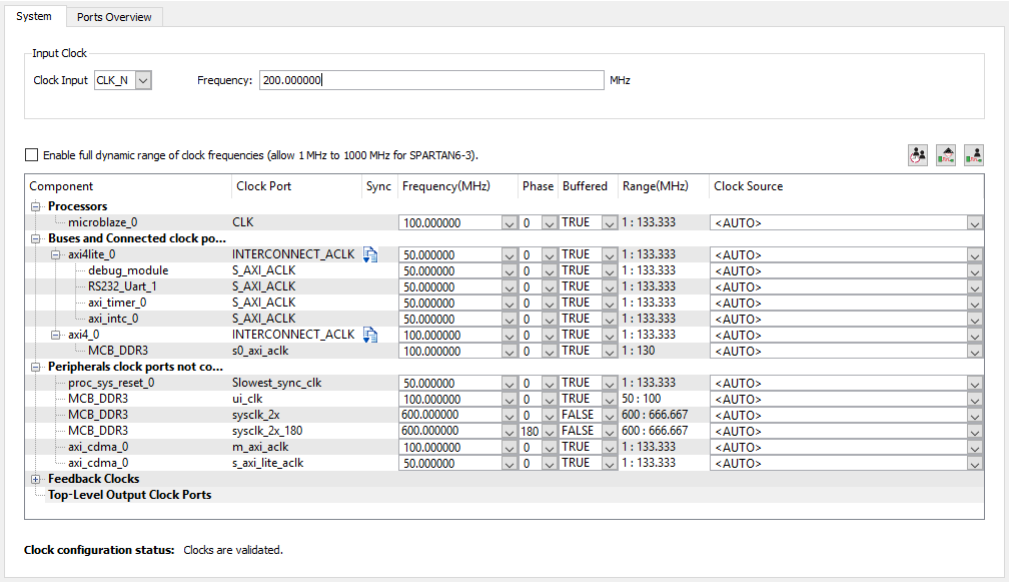

Figure 4.12: XPS Clock Wizard

Before bitstream generation, the developer can go to the Clock Wizard in Project Tab and check all clock constrains required. XPS automatically fills the different clock ports and frequencies of each component, but it is good pratice that the developer validates them (Figure 4.12).

XPS Synthesis Summary (estimated values)					[-]
Report	**Generated**	**Flip Flops Used**	**LUTs Used**	**BRAMS Used**	**Errors**
system	qua 21. out 22:51:56 2015	5415	5772	37	0
system_axi_intc_0_wrapper	qua 21. out 22:50:47 2015	56	90		0
system_axi_timer_0_wrapper	qua 21. out 22:50:35 2015	157	207		0
system_axi_cdma_0_wrapper	qua 21. out 22:50:21 2015	1406	1328	3	0
system_mcb_ddr3_wrapper	qua 21. out 22:47:34 2015	452	840		0
system_rs232_uart_1_wrapper	qua 21. out 22:47:12 2015	85	103		0
system_axi4_0_wrapper	qua 21. out 22:47:01 2015	637	552		0
system_axi4lite_0_wrapper	qua 21. out 22:46:45 2015	313	345		0
system_clock_generator_0_wrapper	qua 21. out 22:46:30 2015				0
system_debug_module_wrapper	qua 21. out 22:46:24 2015	131	142		0
system_microblaze_0_wrapper	qua 21. out 22:46:12 2015	2103	2098	18	0
system_microblaze_0_bram_block_wrapper	qua 21. out 22:45:15 2015			16	0
system_microblaze_0_d_bram_ctrl_wrapper	qua 21. out 22:45:05 2015	2	6		0
system_microblaze_0_dlmb_wrapper	qua 21. out 22:44:58 2015	1			0
system_microblaze_0_i_bram_ctrl_wrapper	qua 21. out 22:44:51 2015	2	6		0
system_microblaze_0_ilmb_wrapper	qua 21. out 22:44:44 2015	1			0
system_proc_sys_reset_0_wrapper	qua 21. out 22:44:35 2015	69	55		0

Figure 4.13: XPS Synthesis Summary (estimated values)

Once the implementation of the project is finished, the developer needs to go through the reports to see if this implementation met expectations and decide what to do if it did not. The Design Summary in XPS allows to quickly access design overview information, messages and reports. It is in this section that the developer can find the XPS reports after generating the netlist from the various source files and also after bitstream generation. It presents the device utilization summary, which shows the estimated utilization (Figure 4.13) after XST synthesis and the actual utilization after mapping (Table 4.1).

Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	4,294	54,576	7%
Number used as Flip Flops	4,284		
Number used as Latches	0		
Number used as Latch-thrus	0		
Number used as AND/OR logics	10		
Number of Slice LUTs	4,665	27,288	17%
Number used as logic	4,163	27,288	15%
Number using O6 output only	3,124		
Number using O5 output only	90		
Number using O5 and O6	949		
Number used as ROM	0		
Number used as Memory	287	6,408	4%
Number used as Dual Port RAM	88		
Number using O6 output only	4		
Number using O5 output only	0		
Number using O5 and O6	84		
Number used as Single Port RAM	0		
Number used as Shift Register	199		
Number using O6 output only	60		
Number using O5 output only	1		
Number using O5 and O6	138		
Number used exclusively as route-thrus	215		
Number with same-slice register load	190		
Number with same-slice carry load	25		
Number with other load	0		
Number of occupied Slices	1,921	6,822	28%
Number of MUXCYs used	780	13,644	5%
Number of LUT Flip Flop pairs used	5,641		
Number with an unused Flip Flop	1,841	5,641	32%
Number with an unused LUT	976	5,641	17%
Number of fully used LUT-FF pairs	2,824	5,641	50%
Number of unique control sets	306		
Number of slice register sites lost to control set restrictions to control set restrictions	1,199	54,576	2%

Table 4.1: Device Utilization Summary (actual values)

4.4 Software Development Kit

As described in the SDK application development in section 4.2, the creation of a Board Support Package that contains a library of routines for the application use, is the first step. When making the BSP, SDK asks the developer for the target hardware device and the operating system to be used in the project. Therefore, the exported files from the XPS were selected to specify the target device and, due to the reasons already stated in section 4.2, Standalone operating system was chosen. These selections are followed by the Board Support Package settings that provides extra support libraries selection (no extra selection was made), ports for the standard input/output stream (RS2323_uart selected), Profiling that will be used and explained after, drivers selection to each component (were left as default) and configuration for cpu driver that will be also changed due to profiling. The next step is Application Project creation, where target Hardware files were selected, OS Platform set as Standalone and programming language was selected as C++.

Then, the linker script needs to be set as can be seen in figure 4.14. A Linker script controls the linking process and it is required if the design contains a discontinuous memory space. It is responsible for mapping the code and data to a specified memory space, setting the entry point to the executable and reserving space for the heap and stack. SDK allows the developer to describe all memory layout of the target machine and specify where each section of the program should be placed in memory. The Linker script is used, as previously said, by the linker to perform the final step for the creation of an executable.

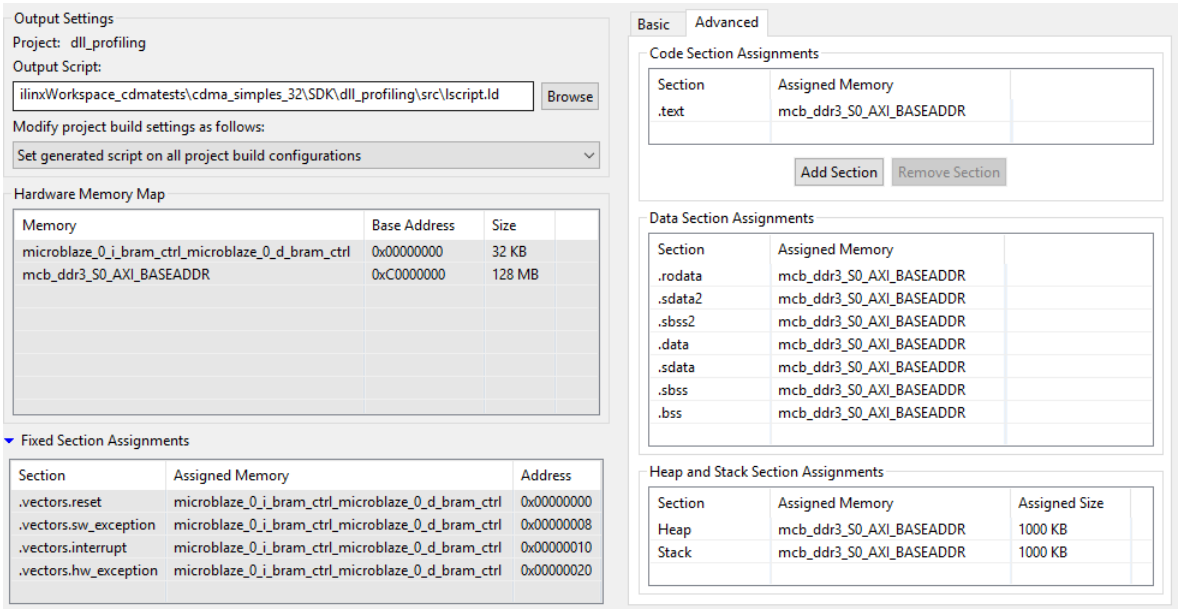

Figure 4.14: SDK Linker Script

Linker Script editing allows the change size of each executable file sections according to table 4.2, as well as, the selection of the memory region to be used (DDR3 or BRAM are the available ones in this case study).

.text	Text section
.rodata	Read-only data section
.sdata2	Small read-only data section
.sbss2	Small read-only uninitialized data section
.data	Read-write data section
.sdata	Small read-write data section
.sbss	Small uninitialized data section
.bss	Uninitialized data section

Table 4.2: Sectional layout of an executable file

To debug, run, and profile an application, a configuration that captures the settings for executing the application, must be created. The configurations for debugging, running, and profiling an application are similar; differences between them will be explained below. It is important to regard that to profile an application, Run configuration must be created [68].

The SDK debugger enables the developer to see what is happening to a program while it executes. A set of breakpoints or watchpoints to stop the processor can be set to achieve a step through program execution, to view the program variables and stack, and to view the contents of the memory in the system. The SDK debugger uses the GNU Debugger (GDB) with Xilinx Microprocessor Debugger (XMD) as the underlying debug engine. It translates each user interface action into a sequence of GDB commands and processes the output from GDB to display the current state of the program being debugged. It communicates to the processor on the hardware and Instruction Set Simulator (ISS) target using XMD [68].

The developer can also run software applications on a hardware target using SDK. The program will run up to termination or when expressed command from the developer, that can be made at any time of execution. The run workflow is similar to debugging with the exception that the executable is performed by the SDK Run Console instead of the SDK Debug Perspective that is used for debugging [68].

Xilinx SDK includes profiling tools that help to identify bottle necks that might occur in the developed code due to the interaction of functions that are executed within the programmable logic and functions executed on the processor [69]. SDK supports hierarchical profiling, allowing the developer to view which called functions, or which calling functions are affecting processor performance the most. In order to use Profiling, enable_sw_intrusive_profiling field should be set to true and the handed over timer needs to be selected for its usage by the profile libraries. Furthermore, BSP also requires to have the -pg flag added in the extra_compiler_flags option of the cpu compiler. Besides setting up the BSP, software application code modification to enable interrupts is required. Then, profile run configuration needs to be done in the Profiler Options tab of the Run Configurations as shown in figure 4.15.

The sampling frequency determines the frequency of which timer interrupts are generated. When set to a higher frequency, more samples are obtained which provides more accuracy, but it is highly software-intrusive due to the number of calls that are inserted to collect data. Therefore, it is crucial to choose sampling frequency according to a compromise between accuracy and program delay, which increases with the sampling frequency.

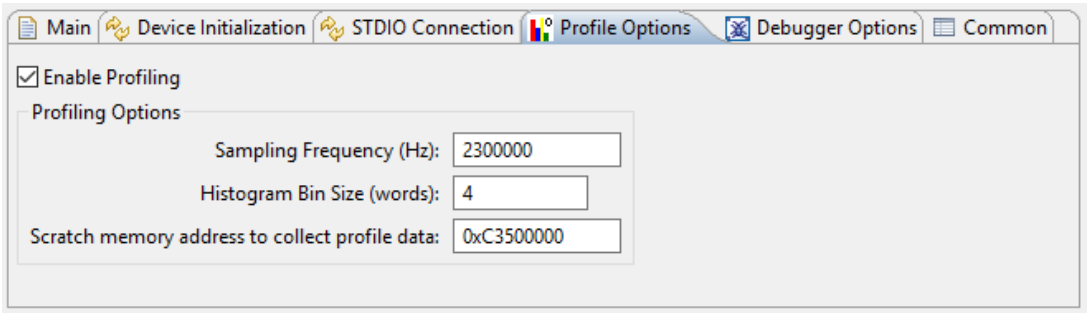

Figure 4.15: SDK Profile Options

The program text region is divided into multiple bins. When a program is interrupted because of the sampling frequency, the Histogram Bin size sets the counter value at which Program Counter (PC) value is incremented and determiness how accurate the PC location is in the sample. When a smaller bin size is set, more accurate samples can be achieved. However, using a smaller bin size requires a large number of bins to cover the entire text region, so a large amount of memory space is required for storing the profile data. On the other hand, if a larger bin size is set, more difficulty to identify specific text regions occurs and leads to inaccurate PC detection.

A valid system address that is not used by the software application should be set in Scratch memory field, value which indicates where in memory the profile data must be stored.

When profiling the program on a hardware target, SDK uses Xilinx Microprocessor Debugger (XMD) for communication to the processor using a JTAG interface on the board.

Once the application finishes running (reaches exit), or when the stop button is pressed to stop the program, SDK downloads the profile data and stores it in a file named gmon.out. To view the profile results, double click on the gmon.out file associated with the application in the Project Explorer view. The Xilinx Profiling view in SDK presents a view of the profiling results to visualize the statistics for the profiled Executable and Linkable Format (ELF) file. Profiling results are composed by: Name which is the function name and file name where it resides; Samples that provide the number of times the profile timer interrupt handler detected that the program executed the corresponding function; Calls which is the number of calls made to the corresponding function; Time/Call that gives time usage per function call invocation; and %Time which is the amount of time spent in a function. In order to validate profiling results, an application using the AXI Timer to measure the functions time was developed. Results will be shown in Chapter 5.

4.5 DLL code overview

The previous discussion on XPS and SDK enables the reader with the steps required to understand embedded system implementation. This is required to develop applications with HLS in SDK. The SDK HLS accelerates IP creation by enabling C/C++ specifications to be directly targeted into Xilinx devices without the need to manually create RTL with High Description Languages (HDL)s (VHDL or Verilog). SDK HLS is a major tool for reducing the application development cycle and is able to achieve FPGA resource utilization levels comparable to hand-written RTL code, as said in the Berkeley Design Technology independent evaluation [70]. Based on this analysis, it is believed that the productivity of the proposed

work could be highly increase with HLS tools usage, and therefore C++ code was chosen to be used.

The considerations on the decision of C++ programming over C were numerous. The object oriented programming language C++ is often considered to be a superset of C, which is not strictly true because there are a few differences that can cause some valid C code to be invalid or behave differently in C++. The chosen C++ language has: a much richer standard library with a stronger type checking (use of classes and inheritance); a safe linkage that assures no accidentally call of one routine with wrong type or arguments from another; complex data type with all standard arithmetic operations implemented as operators; a more logically organised code with necessary declarations not requiring to be at the top; a modular programming with classes that promote code reuse and data scope limiting (functions and variables protection with private and protected keywords).

The forthcoming discussion implies that the reader acknowledges the presence of the performed C++ code for the DLL functions in Appendix B. The mentioned C++ code of this work was structured in four classes called DLL, DLLFrame, FIFO and FPGA. The DLL class contains all DLL functions that were discussed in Section 3.4, DLLFrame contains all DLL frame parameters mentioned in Section 3.3, FIFO accommodates all functions for Higher Layer requests attendance and FPGA has all the required driver Application Programming Interface (API) functions for the CDMA, Timer and Interrupt Controller usage. This code division was performed to provide a cleaner and less error prone programming that also provides the ability to test every phase of the processing independently, leading to a more accurate code debugging. This division also envisions to implement data encapsulation, meaning that the implementation details of a class are kept hidden from the remaining ones. This way, classes can only perform a restricted set of operations on the hidden members of the other classes by executing special functions, usually called methods, that were carefully chosen not be overly flexible or too restrictive. The advantage of using this data encapsulation comes when the implementation of the class changes but the interface remains the same and this is achieved through the use of the public, protected and private keywords which are placed in the declaration of the class. Anything in the class placed after the public keyword is accessible to all the users of the class; elements placed after the protected keyword are accessible only to the methods of the class or classes derived from that class; elements placed after the private keyword are accessible only to the methods of the class. However, the four classes division main reason was to enable DLL code usage in other FPGA designs and models or even other hardware by simply modifying the driver functions used on the FPGA class, providing an easy and fast way to adapt the proposed work to any needs.

Now that code structure was introduced, a brief overview highlighting the used SDK key concepts will be reviewed.

The mentioned Xilinx EDK software libraries comprise the generated libraries and BSP files for a given project. The XPS output file XML is used by the Library Generator (LibGen) tool to configure these libraries, device drivers, file systems and interrupt handlers for the embedded processor system. Then, taking libraries and drivers, SDK is able to compile the desired application by including them into a Executable Linked Format (ELF) files that are ready to run on the processor hardware platform.

The include file, xparameters.h, is one of the files created through LibGen and it holds important parameters for each device in the embedded system (base and high addresses of the peripherals and peripheral IDs required by the drivers and user programs). These parameters are needed by the application code software, written in C++ language, to be able

to communicate with the devices in the embedded system, so its reading was mandatory.

In order to develop application code software, the full description of the features of the AXI CDMA, AXI Timer and Interrupt Controller is provided for consultation by the SDK. This can be found in *xaxicdma.h* for the AXI CDMA, *xtmrctr_l.h* for AXI Timer and *xintc.h* for the Interrupt Controller, or by simply going to the BSP Documentation folder present in the project directory that has the web-page links for the Xilinx Processor IP Library to each of them.

As can be seen in the FPGA.cpp part of the Appendix B, the CDMA was configured for simple DMA transfers without interruptions. However, the hardware supports another type of transfer called the Scatter Gather (SG) that requires setting up a Buffer Descriptor (BD), which keeps the transfer information and is updated by the hardware (it provides queuing of multiple transfers). Using the simple DMA transfer has the advantage of ease of use (no need to manage the memory for the buffer descriptors) and for an individual DMA transfer, simple DMA transfer is also faster because of simplicity in software and hardware. Simple DMA transfer was also chosen because it is sufficient in the present design where DMA is used exclusively by this application and one DMA transfer at a time is enough to send the whole frame. A simple DMA transfer only needs the source buffer address, the destination buffer address and the transfer length to perform the DMA transfer, and only one transfer can be submitted to the hardware each time.

The header file, *xacicdma.h*, contains all the features of the AXI CDMA engine and the functions used to access the engine in the embedded system. In the FPGA.cpp file the following functions were used: *XAxiCdma_IsBusy* to check whether or not the hardware is doing a transfer; *XAxiCdma_CfgInitialize* to initialize the CDMA; *XAxiCdma_ResetIsDone* to check whether the hardware reset was done; *Xil_DCacheInvalidateRange* to prevent cache inconsistency (invalidates receive buffer range before passing it to the hardware) and *XAxiCdma_SimpleTransfer* to perform one simple transfer submission. Since interrupt is enabled, a call back function is registered to signal the transfer completion. The *XAxiCdma_SimpleTransfer* function needs the pointer to the driver instance, the address of the source buffer, the address of the destination buffer and the length of the transfer as required arguments. Since CDMA without interruptions was used the callback function and the callback reference pointer arguments are not required and were filled as NULL. It is important to regard that in order to set CDMA properly, addresses and transfer size are required to be aligned to 4Bytes. This is why after changing pointers or transfer_size variables their values are checked to see if they do not fullfill the condition $if(size\%4 != 0)$ and if they do not, their value needs to be adjusted to assure memory alignment.

The used timer counter can operate in two primary modes: compare and capture. In either mode the timer counter may count up or down, being that up is the default. As can be seen in Appendix B, the timer was configured in compared mode with auto reload such that the timer counter will reload itself (with the compare value) automatically and continue repeatedly. The timer functions present in *xtmrctr_l.h* that were used are: *XTmrCtr_Start* to start the specified timer counter; *XTmrCtr_Stop* to stop the timer counter by disabling it; *XTmrCtr_ReadReg* to read the register contents and *XTmrCtr_Reset* to reset the specified timer counter.

In order to use Timer with interrupts, the Interrupt Controller needs to be configured. The required functions that were used are: *XIntc_Initialize* to initialize a specific interrupt controller instance/driver; *XIntc_Connect* to make the connection between the Id of the interrupt source and the associated handler; *XIntc_SetOptions* to set the options for the interrupt

controller driver; *XIntc_Start* to start the interrupt controller by enabling the output from the controller to the processor; *XIntc_Enable* to enable the interrupt source provided as the argument Id and *XIntc_Acknowledge* to acknowledge the interrupt source provided as the argument Id. Each interrupt connected to the Interrupt controller has a unique Interrupt vector address that the processor jumps to for servicing that particular interrupt. In the used normal interrupt mode, the interrupt vector addresses are determined by the software drivers application. Setting up interrupts is largely driver-API dependent, but the general procedure is the same: 1)Use the peripheral API to initialize the hardware if an initialize function is provided; 2)Use the peripheral API to customize the hardware; 3)Register the Handler with *XIntc_RegisterHandler* 4)Master enable the interrupt controller with *XIntc_mMasterEnable*; 5)Enable the individual interrupts with the interrupt controller using *Xintc_mEnableIntr*; 6)Use the peripheral API to enable the peripheral interrupt; 7)Create a function with the handler name and program it to do the desired task.

It is also important to regard that the I/O function commands of *xil_printf* were used in the program instead of *printf*, because they have been optimized for embedded systems with limited memory by being much smaller in code size.

All previous discussion enabled the reader with the used functions on the *fpga.cpp* that, in turn, was divided in a *TransferCDMA* to configure the transfer request, a *DataTransferCheck* test function to determine the transfer success, a *StartTimer* to start timer counter, a *StopTimer* to stop timer counter, a *GetCounterValue* to return timer value, a *ResetCounter* to reset timer counter, a *ConfigTimerCDMA* that runs first time FPGA class is called and configures all CDMA and Timer parameters, a *SetupIntrSystem* that is called by *ConfigTimerCDMA* and configures all interruption settings and a *TimerCounterHandler* that is the callback function of Timer. It is important to regard that functions initialized with FPGA class were set private, so no change in configurations could be performed by outside functions. While functions that request data transfer or the timer were set public so DLL class could call them. The header, *dllframe.h*, contains all mentioned frame constants required to perform data encapsulation and fragmentation, while *dllframe.cpp* has a function for junk frame creation (*initjunkframe*).

A very important key concept taken into account in its study was the use of interruptions. AXI Timer interruption configuration was performed, because Junk Frame is required to be sent before a predefined time (to maintain the lighting functionality of the VLC transceiver (in cases of no DC-OFDM usage) and to fullfill the project asynchronous architecture criteria of buffer fill levels). On the other hand, CDMA was not configured to benefit from interruptions, due to its usage in Simple Mode only together with functions order. The proposed DLL functions were organized in a such way that a transfer of the CDMA will be preceded and proceeded by a processing function, allowing transfers to occur while frame processing is ongoing. With this strategy and with this strategy only, the interruptions do not bring extra value. As will be further discussed, CDMA would benefit from interruptions in cases of Scatter Gather mode usage, allowing the CDMA to store multiple requests instead of only one.

The class FIFO is responsible for Higher Layer requests (HDR and MDR) management. It has functions that store a request (*In*) , remove a request (*Out*) , change a request (*Change*), return request value without removing it from buffer (*Check*), delete all requests (*Delete*) and check the request buffer state (*isEmpty* and *isFull*). To provide data encapsulation from the DLL class, that is the one using this class, FIFO class has all the previously mentioned functions set as public and requires the *push*, *pop*, *change* and *check* private functions to perform the required tasks without risking buffer exposure.

All DLL class main functions behaviour was already discussed in section 3.4, but in order to make the link with the remaining classes, a set of extra functions were required to be programmed. These extra functions are *test_mode*, *Images_Request*, *Data_Request* and *dll_check* that are test functions used for the program debug. The first one is responsible for changing the emitter output buffer addresses to the receiver input buffer address, since DLL was tested separately from the remaining VLCLighting blocks. *HDR_Request* and *MDR_Request* were introduced to simulate Higher Layer requests input, because, again, the DLL was tested separately from remaining VLCLighting blocks. By its turn, *dll_check* was a test function perform to check DLL success by comparing the emitter and receiver frames.

Chapter 5

Experimental validation

In order to accomplish all the following DLL results, three embedded designs were performed in XPS. In each of these embedded designs, CDMA, MIG and AXI Interconnect were configured with matching sizes with 32, 64 and 128 bits. The first results performed are from the CDMA. CDMA results show which of the three designs is the best for the proposed DLL.

5.1 Central Direct Memory Access

As stated, in order to test the proposed DLL design, a set of XPS designs were synthesized with different CDMA data transfer widths, different DDR3 data port settings and different AXI Interconnect data widths. In these design synthesis it was observed that CDMA and AXI Interconnect allow data transfer widths from 32 to 1024 bits, more specifically of 32, 64, 128, 256, 512 and 1024 bits. However, the Xilinx Memory Interface Generator (MIG) that is responsible for generating memory interfaces for the Xilinx FPGAs (in this study is responsible for generating the DDR3 interface for the Spartan 605) only accepts 32, 64 and 128 bits of port configuration values. Taken this into account, the CDMA and AXI Interconnect were configured together with MIG in three XPS designs of 32, 64 and 128 bits of data width, that will be addressed in this chapter as 32, 64 and 128-DLL configurations. Another important aspect is that all these three configurations present no issues in the data alignment given that the DLL frame size is of 208 Bytes.

After the creation of the three DLL configurations, it was performed an analysis of the TransferCDMA function time for the DLL frame size of 208 Bytes (Table 5.1). It is important to regard that results shown contain the time for the configuration, the transfer, the validation and invalidation of memory, but can be the comparison basis between each embedded design. The results in this Table were performed with a Timer at a 50 MHz frequency. A total of 5 samples of the timer counter were taken to each case to achieve the best average value of the transfer time. This means that to calculate the transfer time, the average timer samples taken need to be multiplied by 20 ns.

Sample number	CDMA data width		
	32	64	128
#1	2356	434	533
#2	2211	424	533
#3	2195	407	533
#4	2211	411	534
#5	2206	410	533
Average:	2236	418	534
Time (μs):	40	8	10

Table 5.1: Timer samples comparison between different CDMA data widths

As can be seen in Table 5.1, 64-DLL outperforms 32-DLL by a factor of more than five, while 128-DLL design shows, as expected, a decrease in DLL frame transfer time. The efficiency decrease from 64-DLL to 128-DLL was expected due to Spartan 605 Evaluation Kit limitations in the XPS design. After reading Micron MT41J64M16LA-187E Datasheet [71], it can be seen that this model has 16 data ports that are able to perform a 16bits read/write operation each DDR3 clock. Since the MIG clock is limited to 600MHz in the XPS design, the transfer of 128bits is not possible to be made in one clock cycle of the AXI bus that has a clock frequency of 100MHz. In order to perform 128bits transfer with the DDR3 at the 100MHz bus frequency, a 800MHz clock frequency configuration is required to the MIG. Therefore, the 128-DLL choice is proven to be inefficient, since a 128bit transfer cannot be completed in one AXI bus clock. Given that 64-DLL outperforms the remaining two, it was the design used in the DLL study and will be the one addressed further on.

Other results considering only the CDMA transfer time for 64-DLL case show a counter of 42 samples. This result shows that the transfer itself takes $0.84\mu s$, which is a fraction of the required time for the transfer. With this result in mind, it can be seen that the maximum CDMA throughput will be 1,980 Gbits/sec.

5.2 Profiling

As previously stated, this work main focus was to design an efficient DLL, specifically in terms of throughput. In order to measure throughput, accurate time measurements are vital so, logically, part of this chapter is dedicated to the discussion of the results obtained with the available time measurement tools. In this section, software profiling (discussed in Section 4.4) parameters choosing, as well as, its comparison with timer measurements will be addressed.

The profiling configuration is achieved with two main parameters, the profiling frequency and the Bin size. Tests with a fixed Bin size and different profiling frequencies can be seen in Tables 5.2 and 5.3. These tests were performed in the 64-DLL architecture and they show a great variation in the measured Time/Call of the TransferCDMA function that is responsible for transferring a DLL frame. So, in order to understand the profiling results inconsistency, graphs 5.1 and 5.2 that show the Sampling Frequency affinity with the transfer function time, were plotted.

Profiling Frequency (MHz)	Total Samples	Sample Number	Sample counts as (ns)	Calls	Time/Call (μs)
2	10145	97	500	3	11
2,1	8896	98	476	3	15,55
2,2	11345	130	454	3	19,69
2,25	11604	146	444	3	22,12
2,3	12357	144	434	3	20,87
2,4	12329	133	416	3	18,47
2,5	12334	22	400	3	8,8
3	12600	169	333	3	18,77
3,5	12540	168	285	3	16

Table 5.2: Profiling of TransferCDMA function with 4words (16bytes) Bin size

Profiling Frequency (MHz)	Total Samples	Sample Number	Sample counts as (ns)	Calls	Time/Call (μs)
2	10092	91	500	3	15,16
2,1	8896	98	476	3	15,55
2,2	11321	117	454	3	17,72
2,25	11521	140	444	3	20,74
2,3	12269	137	434	3	19,85
2,4	12355	150	416	3	20,83
2,5	12437	150	400	3	20
3	12502	153	333	3	17
3,5	12582	175	285	3	16,66

Table 5.3: Profiling of TransferCDMA function with 2words (8bytes) Bin size

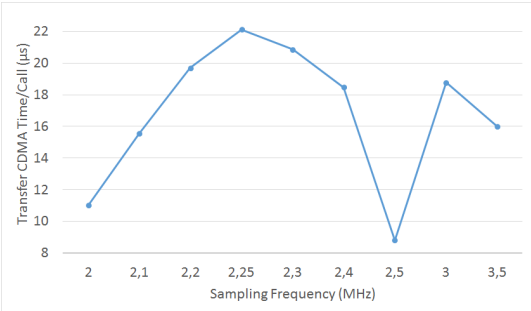

Figure 5.1: 4 Bin Words(16bytes)

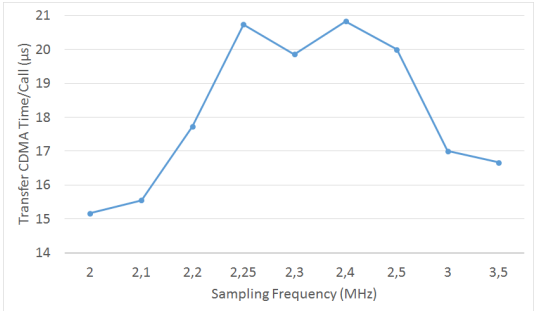

Figure 5.2: 2 Bin Words(8bytes)

As can be seen in Table 5.2, the measured time by the profiling varies between 8,8μs and 22,12μs. An identical variation of the measured values can be seen in Table 5.3 with 16,66μs and 20,84μs. As expected, all profiling results are higher than the 8μs measured with timer (Table 5.1). This is due to the software intrusive nature of the profiling that requires the program to be periodically interrupted to obtain a sample of its program counter location and store the profile information in the memory. Since the achieved profiling results present

big discrepancies between different Profiling frequencies and the ones obtained with timer, the timer will be the measurement tool used in the DLL tests.

5.3 Higher Layer services requests

To prove the functional behaviour and show the reader the DLL steps made to perform the attendance of the higher layer requests, this section will present DLL buffers data that can be seen in the Debug mode. In order to be easier to understand, all data was highlighted according to its meaning. The HDR request data was highlighted on blue, MDR request data was highlighted on green and the header was highlighted on red.

((this)->hdr_in).ptr : 0xC4000000 <Hex> | New Renderings...

Address	0 - 3	4 - 7	8 - B	C - F
C4000000	409C0000	419C0000	429C0000	439C0000
C4000010	449C0000	459C0000	469C0000	479C0000
C4000020	489C0000	499C0000	4A9C0000	4B9C0000
C4000030	4C9C0000	4D9C0000	4E9C0000	4F9C0000
C4000040	509C0000	519C0000	529C0000	539C0000
C4000050	549C0000	559C0000	569C0000	579C0000
C4000060	589C0000	599C0000	5A9C0000	5B9C0000
C4000070	5C9C0000	5D9C0000	5E9C0000	5F9C0000
C4000080	609C0000	619C0000	629C0000	639C0000
C4000090	649C0000	659C0000	669C0000	679C0000
C40000A0	689C0000	699C0000	6A9C0000	6B9C0000
C40000B0	6C9C0000	6D9C0000	6E9C0000	F017F012
C40000C0	3117B013	F0137017	F417F017	C895E415
C40000D0	F017F257	30037003	E0146014	E017F017
C40000E0	F0063003	7017F217	F015F017	C814F415

Figure 5.3: HDR Input Buffer

((this)->mdr_in).ptr : 0xC3000000 <Hex> | New Renderings...

Address	0 - 3	4 - 7	8 - B	C - F
C3000000	88130000	89130000	8A130000	80175015
C3000010	F217D017	70133017	D016D016	D017E005
C3000020	B1033007	7007F013	F417D014	C816F054
C3000030	7203E017	32073313	D016E094	F017F016
C3000040	3003B003	B017B013	D017F017	C057C416
C3000050	F21FF017	B0033007	E095F017	C057F055
C3000060	32077017	F213B013	F10CF117	C016D816
C3000070	B217F01F	B013B203	C444C056	B415E895
C3000080	70273213	B01FB007	F804D057	CC97F014
C3000090	F017F01B	B2133007	F017F016	F016D017
C30000A0	B103F03F	7817B005	C057F017	E815C016
C30000B0	7003B017	72137013	F817E016	F415F057
C30000C0	B20F3203	F0177007	E817E417	D014C017
C30000D0	7217F013	F0273017	C014F015	F017FC97
C30000E0	301B3003	701FF03B	F857D017	C016F014

Figure 5.4: MDR Input Buffer

((this)->frame).tx_e : 0xC0635000 <Hex> | New Renderings...

Address	0 - 3	4 - 7	8 - B	C - F
C0635000	20060420	10002000	409C0000	419C0000
C0635010	429C0000	439C0000	449C0000	459C0000
C0635020	469C0000	479C0000	489C0000	499C0000
C0635030	4A9C0000	4B9C0000	4C9C0000	4D9C0000
C0635040	4E9C0000	4F9C0000	509C0000	519C0000
C0635050	529C0000	539C0000	549C0000	559C0000
C0635060	569C0000	579C0000	589C0000	599C0000
C0635070	5A9C0000	5B9C0000	5C9C0000	5D9C0000
C0635080	5E9C0000	5F9C0000	609C0000	619C0000
C0635090	629C0000	639C0000	649C0000	659C0000
C06350A0	669C0000	679C0000	689C0000	699C0000
C06350B0	6A9C0000	6B9C0000	6C9C0000	6D9C0000
C06350C0	6E9C0000	F017F012	88130000	89130000
C06350D0	0FE802E8	03280728	0CE84CC0	0DEA8EA8
C06350E0	036823A8	2FA803E9	05E04FE0	0CC40CC4
C06350F0	07E80BA8	020A0360	0CE80CE0	47E00DF8

Figure 5.5: DLL Emitter internal Buffer

((this)->frame).rx_e : 0xC0639000 <Hex> | New Renderings...

Address	0 - 3	4 - 7	8 - B	C - F
C0639000	20060420	10002000	409C0000	419C0000
C0639010	429C0000	439C0000	449C0000	459C0000
C0639020	469C0000	479C0000	489C0000	499C0000
C0639030	4A9C0000	4B9C0000	4C9C0000	4D9C0000
C0639040	4E9C0000	4F9C0000	509C0000	519C0000
C0639050	529C0000	539C0000	549C0000	559C0000
C0639060	569C0000	579C0000	589C0000	599C0000
C0639070	5A9C0000	5B9C0000	5C9C0000	5D9C0000
C0639080	5E9C0000	5F9C0000	609C0000	619C0000
C0639090	629C0000	639C0000	649C0000	659C0000
C06390A0	669C0000	679C0000	689C0000	699C0000
C06390B0	6A9C0000	6B9C0000	6C9C0000	6D9C0000
C06390C0	6E9C0000	F017F012	88130000	89130000
C06390D0	0FE802E8	03680728	0CE84CC0	0DEA8EA8
C06390E0	036823A8	2FA803E9	05E04F60	0CC40CC0
C06390F0	07E80BA8	020A0360	0CE80EE0	C7E00DE8

Figure 5.6: Output Buffer DLL Emitter

((this)->frame).tx_r : 0xC0639000 <Hex> | New Renderings...

Address	0 - 3	4 - 7	8 - B	C - F
C0639000	20060420	10002000	409C0000	419C0000
C0639010	429C0000	439C0000	449C0000	459C0000
C0639020	469C0000	479C0000	489C0000	499C0000
C0639030	4A9C0000	4B9C0000	4C9C0000	4D9C0000
C0639040	4E9C0000	4F9C0000	509C0000	519C0000
C0639050	529C0000	539C0000	549C0000	559C0000
C0639060	569C0000	579C0000	589C0000	599C0000
C0639070	5A9C0000	5B9C0000	5C9C0000	5D9C0000
C0639080	5E9C0000	5F9C0000	609C0000	619C0000
C0639090	629C0000	639C0000	649C0000	659C0000
C06390A0	669C0000	679C0000	689C0000	699C0000
C06390B0	6A9C0000	6B9C0000	6C9C0000	6D9C0000
C06390C0	6E9C0000	F017F012	88130000	89130000
C06390D0	0FE802E8	03680728	0CE84CC0	0DEA8EA8
C06390E0	036823A8	2FA803E9	05E04F60	0CC40CC0
C06390F0	07E80BA8	020A0360	0CE80EE0	C7E00DE8

Figure 5.7: DLL Receiver internal Buffer

((this)->frame).rx_r : 0xC0839000 <Hex> | New Renderings...

Address	0 - 3	4 - 7	8 - B	C - F
C0839000	20060420	10002000	409C0000	419C0000
C0839010	429C0000	439C0000	449C0000	459C0000
C0839020	469C0000	479C0000	489C0000	499C0000
C0839030	4A9C0000	4B9C0000	4C9C0000	4D9C0000
C0839040	4E9C0000	4F9C0000	509C0000	519C0000
C0839050	529C0000	539C0000	549C0000	559C0000
C0839060	569C0000	579C0000	589C0000	599C0000
C0839070	5A9C0000	5B9C0000	5C9C0000	5D9C0000
C0839080	5E9C0000	5F9C0000	609C0000	619C0000
C0839090	629C0000	639C0000	649C0000	659C0000
C08390A0	669C0000	679C0000	689C0000	699C0000
C08390B0	6A9C0000	6B9C0000	6C9C0000	6D9C0000
C08390C0	6E9C0000	F017F012	88130000	89130000
C08390D0	F017F017	71037003	D417E006	F017F015
C08390E0	33233001	F117F117	EC97F094	C814E014
C08390F0	F303F01F	7003F012	D015E014	F017F015

Figure 5.8: Output Buffer DLL Receiver

((this)->hdr_out).ptr : 0xC1000000 <Hex> | New Renderings...

Address	0 - 3	4 - 7	8 - B	C - F
C1000000	409C0000	419C0000	429C0000	439C0000
C1000010	449C0000	459C0000	469C0000	479C0000
C1000020	489C0000	499C0000	4A9C0000	4B9C0000
C1000030	4C9C0000	4D9C0000	4E9C0000	4F9C0000
C1000040	509C0000	519C0000	529C0000	539C0000
C1000050	549C0000	559C0000	569C0000	579C0000
C1000060	589C0000	599C0000	5A9C0000	5B9C0000
C1000070	5C9C0000	5D9C0000	5E9C0000	5F9C0000
C1000080	609C0000	619C0000	629C0000	639C0000
C1000090	649C0000	659C0000	669C0000	679C0000
C10000A0	689C0000	699C0000	6A9C0000	6B9C0000
C10000B0	6C9C0000	6D9C0000	6E9C0000	F017F012
C10000C0	B017A017	70177117	E007F017	D014C855
C10000D0	F017F007	B02F3013	E017C014	D015C857
C10000E0	70177003	D01FF01F	F017F01F	C014F014

Figure 5.9: HDR Output Buffer

((this)->mdr_out).ptr : 0xC2000000 <Hex> | New Renderings...

Address	0 - 3	4 - 7	8 - B	C - F
C2000000	88130000	89130000	F015F417	C055E055
C2000010	B017F003	32133017	D014F496	F0975057
C2000020	3017B013	F017F01F	F496F013	C054F016
C2000030	F117B037	30137013	D014D016	D017F013
C2000040	300A701F	7137F217	B056F017	C054C014
C2000050	6007F013	31033203	D096C057	F014D015
C2000060	300B7007	B01FB317	C015F017	C014F091
C2000070	F0136037	B01FB003	D015D015	F013F817
C2000080	B0123207	71177013	D015F017	F017C815
C2000090	7016F013	3213F00B	C011D016	F017F814
C20000A0	F2037203	F817F117	F817C017	10D4C017
C20000B0	F017F017	31032017	C014C054	F015D016
C20000C0	B01F7103	C117F003	E017D015	E094C856
C20000D0	7213B113	7007702F	C014F016	E017F415
C20000E0	30033207	F0177017	F015F097	F015D017

Figure 5.10: MDR Output Buffer

Figure 5.3 represents the DLL emitter input buffer of the HDR requests, while Figure 5.9 the DLL receiver output buffer for those same requests. Similarly, Figure 5.4 shows the DLL emitter input buffer of the MDR requests and Figure 5.10 shows the DLL receiver output buffer of those requests. As can be seen in the referred images, the HDR and MDR requests were correctly transmitted. In order to achieve frame transfer, the correct header according to what was discussed in Section 3.3 needs to be performed. In Figures 5.5, 5.6, 5.7 and 5.8 the calculated header for the MDR Fragmentation case (HDR=192, MDR=16) can be seen, as mentioned, highlighted in red. That same header is fully explained in Figure 5.11 where the reader can see the parameters considered to its calculation and understand the reason for the corresponding values of [0x20 0x06 0x04 0x20 0x10 0x00 0x20 0x00]. It is important to notice that the reserved field is currently not being used, so the 0x20 that can be seen in the header has no meaning in the DLL transfer.

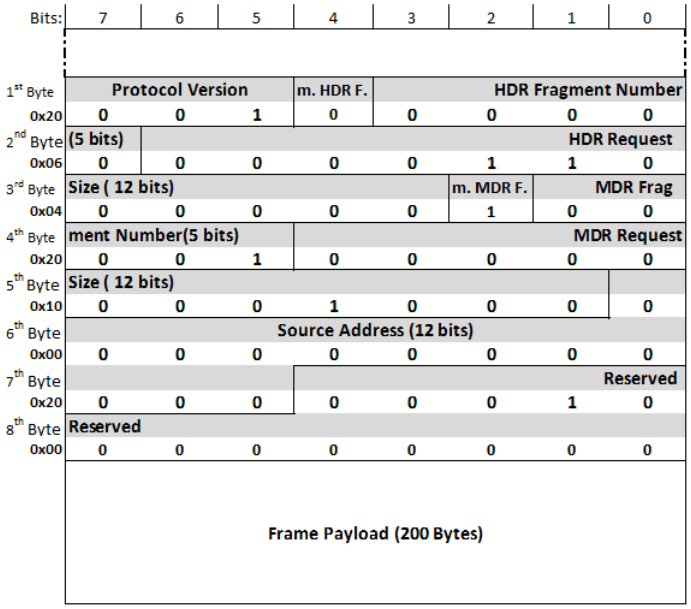

Figure 5.11: Calculated Header MDR Fragmentation

5.4 DLL functions time results

Now that the DLL Emitter and Receiver functioning was illustrated in the previous Section, DLL function time with respective throughput results will be presented. In order to measure the time of the DLL functions regarding all plausible fragmentation cases, four scenarios were sighted in this work, which were no services fragmentation, HDR fragmentation, MDR fragmentation and fragmentation of both services.

5.4.1 No Fragmentation

The first considered case is the no fragmentation of both HDR and MDR requests that can be seen in Tables 5.4 and 5.5. As can be seen in those Tables, five counter (AXI Timer) samples of each function were taken and since DLL code has some extra instructions outside those functions, samples with the total time of the DLL Emitter and DLL Receiver were also included in the respective Tables. Since AXI Timer is known to be 50MHz, the time taken by each function can be calculated by simply multiplying the sampling period ($20ns$) by the number of samples. Furthermore, an average of the five samples of DLL Emitter and DLL Receiver will be used in the respective throughput calculation using Equation 5.1.

Sample number	Fragment Control	Admission Control HDR	Header Coder	Admission Control MDR	Link Management	DLL Emitter
#1	804	570	174	292	398	**2262**
#2	789	568	174	292	398	**2244**
#3	788	569	174	292	398	**2250**
#4	794	569	175	292	406	**2263**
#5	798	569	174	292	398	**2266**
Average:	795	569	175	292	400	**2257**
Time (μs):	15,9	11,38	3,5	5,84	8	**45,14**

Table 5.4: Timer samples of DLL Emitter with no Fragmentation

Sample number	Link Management	Header Decoder	Defragmentation Control	Higher Layer Control	DLL Receiver
#1	366	181	151	609	**1413**
#2	366	179	142	609	**1423**
#3	374	179	142	616	**1423**
#4	372	179	142	609	**1415**
#5	366	179	142	609	**1423**
Average:	369	180	144	611	**1420**
Time (μs):	7,38	3,6	2,88	12,22	**28,4**

Table 5.5: Timer samples of DLL Receiver with no fragmentation

$$Throughput = \frac{Payload\,size(200Bytes) \times 8bits}{DLLTime} \tag{5.1}$$

By considering that 1600 bits are sent in the 45,14 μs, the Emitter Throughput can be calculated (Equation 5.2);

$$Throughput\,Emitter = 35,45Mbits/sec \tag{5.2}$$

With the same reasoning, the Receiver Throughput can be calculated (Equation 5.3).

$$Throughput\,Receiver = 56,34Mbits/sec \tag{5.3}$$

Since the throughput of the DLL Emitter is lower than the one obtained in the DLL Receiver, it will be the one conditioning the total DLL throughput. This reason being that receiver cannot outperform the emitter and the maximum DLL throughput is given by the DLL Emitter throughput.

5.4.2 HDR Fragmentation

After the no fragmentation analysis, the second case is the fragmentation of the HDR request. Since HDR fragmentation occurs with requests larger than 192 Bytes, a 250 Bytes of HDR request was tested and four Tables are presented below to show the two frame construction in Emitter and defragmentation in the Receiver. The first frame will consist of the HDR maximum 192 Bytes and the MDR maximum of 8 Bytes, while the second frame

will have no MDR request and will transmit the remaining 58 Bytes of the HDR request that was fragmented.

Sample number	Fragment Control	Admission Control HDR	Header Coder	Admission Control MDR	Link Management	DLL Emitter
#1	795	572	178	292	302	**2218**
#2	799	567	174	292	302	**2231**
#3	790	563	174	292	302	**2222**
#4	798	566	174	292	302	**2227**
#5	785	565	178	294	302	**2222**
Average:	794	567	176	293	302	**2224**
Time (μs):	15,88	11,34	3,52	5,86	6,04	**44,48**

Table 5.6: Timer samples of DLL Emitter with HDR Fragmentation First Frame

Sample number	Fragment Control	Admission Control HDR	Header Coder	Admission Control MDR	Link Management	DLL Emitter
#1	527	282	138	0	268	**1227**
#2	531	282	138	0	268	**1221**
#3	526	282	138	0	268	**1226**
#4	536	282	138	0	268	**1221**
#5	527	282	138	0	268	**1230**
Average:	530	282	138	0	268	**1225**
Time (μs):	10,6	5,64	2,76	0	5,36	**24,5**

Table 5.7: Timer samples of DLL Emitter with HDR Fragmentation Second Frame

Sample number	Link Management	Header Decoder	Defragmentation Control	Higher Layer Control	DLL Receiver
#1	313	185	147	609	1415
#2	313	180	151	609	1412
#3	313	179	146	609	1419
#4	313	179	151	609	1412
#5	313	189	146	615	1415
Average:	313	183	149	611	1415
Time (μs):	6,26	3,66	2,98	12,22	28,3

Table 5.8: Timer samples of DLL Receiver with HDR fragmentation First Frame

Sample number	Link Management	Header Decoder	Defragmentation Control	Higher Layer Control	DLL Receiver
#1	269	141	92	528	1146
#2	269	141	92	528	1138
#3	269	141	92	528	1145
#4	269	141	92	528	1138
#5	269	141	92	528	1144
Average:	269	141	92	528	1143
Time (μs):	5,38	2,82	1,84	10,56	22,86

Table 5.9: Timer samples of DLL Receiver with HDR fragmentation Second Frame

Since in this scenario the second frame is not full, the DLL throughput is expected to be lower. It is important to remark that this case scenario (HDR Fragmentation) will not be very usual, due to the proper sizing of the DLL frame size to fit one MPEG-TS frame, as previously mentioned. That said, this scenario only poses a mere illustration of the time differences between the DLL functions with and without HDR fragmentation, so the reader can have an idea of the fragmentation impact in the DLL functions time.

In this case, different amount of bits were sent (2064), but the Emitter throughput follow same logic as previous case. The result can be seen in the Equation 5.4. Receiver throughput result is shown in Equation 5.5.

$$Throughput\,Emitter = 23,20 Mbits/sec \tag{5.4}$$

$$Throughput\,Receiver = \frac{2064 bits}{50,86 \mu s} = 31,27 Mbits/sec \tag{5.5}$$

5.4.3 MDR Fragmentation

The third case considered is similar to the last one, but instead of having HDR Fragmentation, it was considered to exist MDR Fragmentation. Again, the first frame sent contains 192 Bytes of HDR request and 8 Bytes of MDR request, while the second frame only contains 8 Bytes of the MDR request.

Sample number	Fragment Control	Admission Control HDR	Header Coder	Admission Control MDR	Link Management	DLL Emitter
#1	860	574	178	292	302	**2326**
#2	865	563	174	292	302	**2323**
#3	867	569	174	292	302	**2329**
#4	875	567	174	292	302	**2331**
#5	876	563	174	292	302	**2337**
Average:	869	568	175	292	302	**2330**
Time (μs):	17,38	11,36	3,5	5,84	6,04	**46,6**

Table 5.10: Timer samples of DLL Emitter with MDR Fragmentation First Frame

Sample number	Fragment Control	Admission Control HDR	Header Coder	Admission Control MDR	Link Management	DLL Emitter
#1	342	0	138	251	268	**996**
#2	352	0	138	251	268	**996**
#3	342	0	138	251	268	**996**
#4	342	0	138	251	268	**996**
#5	342	0	138	251	268	**996**
Average:	344	0	138	251	268	**996**
Time (μs):	6,88	0	2,76	5,02	5,36	**19,92**

Table 5.11: Timer samples of DLL Emitter with MDR Fragmentation Second Frame

Sample number	Link Management	Header Decoder	Defragmentation Control	Higher Layer Control	DLL Receiver
#1	313	181	155	609	**1415**
#2	312	179	146	609	**1416**
#3	312	179	146	609	**1407**
#4	313	183	151	609	**1420**
#5	313	179	155	609	**1414**
Average:	313	181	151	609	**1415**
Time (μs):	6,26	3,62	3,02	12,18	**28,3**

Table 5.12: Timer samples of DLL Receiver with MDR fragmentation First Frame

Sample number	Link Management	Header Decoder	Defragmentation Control	Higher Layer Control	DLL Receiver
#1	269	141	92	515	**1125**
#2	269	141	91	515	**1123**
#3	271	141	92	515	**1126**
#4	269	141	92	515	**1130**
#5	269	141	92	515	**1126**
Average:	270	141	92	515	**1126**
Time (μs):	5,4	2,82	1,84	10,3	**22,52**

Table 5.13: Timer samples of DLL Receiver with MDR fragmentation Second Frame

In this case, a total of 1664 bits were sent in the two frames. Therefore, the Emitter throughput can be calculated as in Equation 5.6.

$$Throughput\,Emitter = 24,05 Mbits/sec \tag{5.6}$$

Since receiver shows a different time result, the Receiver throughput of MDR Fragmentation case can be seen in Equation 5.7.

$$Throughput\,Receiver = 31,48 Mbits/sec \tag{5.7}$$

As expected the throughput shows worst performance when compared to the no fragmentation case. This MDR Fragmentation scenario when compared to the HDR Fragmentation

scenario shows similar performance, which confirms the premiss that the DLL throughput is mainly affected by the processing functions and not the CDMA transfers (in HDR Fragmentation scenario 50 extra Bytes were sent).

5.4.4 HDR and MDR Fragmentation

The fourth and last considered scenario is the fragmentation of both HDR and MDR requests. This test was performed with 250 Bytes of HDR request and 18 Bytes of MDR request, which means that first frame contains HDR=192 Bytes and MDR=8 Bytes and the second frame only contains HDR=58 Bytes and MDR=8 Bytes.

Sample number	Fragment Control	Admission Control HDR	Header Coder	Admission Control MDR	Link Management	DLL Emitter
#1	791	565	174	292	302	**2217**
#2	782	570	174	292	302	**2227**
#3	783	572	174	292	305	**2228**
#4	782	572	174	292	302	**2215**
#5	797	567	176	292	302	**2222**
Average:	787	570	175	292	303	**2222**
Time (μs):	15,74	11,4	3,5	5,84	6,06	**44,44**

Table 5.14: Timer samples of DLL Emitter with HDR and MDR Fragmentation First Frame

Sample number	Fragment Control	Admission Control HDR	Header Coder	Admission Control MDR	Link Management	DLL Emitter
#1	532	282	150	256	268	**1485**
#2	533	282	150	256	268	**1481**
#3	526	282	150	256	268	**1490**
#4	526	282	150	256	268	**1481**
#5	527	282	150	256	268	**1488**
Average:	529	282	150	256	268	**1485**
Time (μs):	10,58	5,64	3	5,12	5,36	**29,7**

Table 5.15: Timer samples of DLL Emitter with HDR and MDR Fragmentation Second Frame

Sample number	Link Management	Header Decoder	Defragmentation Control	Higher Layer Control	DLL Receiver
#1	312	179	146	609	**1416**
#2	321	179	155	609	**1415**
#3	312	179	146	609	**1417**
#4	312	179	146	617	**1420**
#5	313	188	133	609	**1413**
Average:	314	181	146	611	**1417**
Time (μs):	6,28	3,62	2,92	12,22	**28,34**

Table 5.16: Timer samples of DLL Receiver with HDR and MDR fragmentation First Frame

Sample number	Link Management	Header Decoder	Defragmentation Control	Higher Layer Control	DLL Receiver
#1	269	141	92	515	**1125**
#2	269	141	92	515	**1125**
#3	269	141	92	515	**1125**
#4	269	141	92	515	**1125**
#5	269	141	92	515	**1125**
Average:	269	141	92	515	**1125**
Time (μs):	5,38	2,82	1,84	10,3	**22,5**

Table 5.17: Timer samples of DLL Receiver with HDR and MDR fragmentation Second Frame

In this last considered case, a total of 2128 bits were sent in both frames. This size considered with the DLL Emitter time can be used to achieve the throughput in Equation 5.8. On the other hand, the Receiver throughput result can be seen in Equation 5.9.

$$Throughput\,Emitter = 21,58 Mbits/sec \tag{5.8}$$

$$Throughput\,Receiver = 31,47 Mbits/sec \tag{5.9}$$

As can be seen, the DLL Emitter and Receiver Throughput are lower than the remaining two previous fragmentation scenarios. This is due to the extra instructions performed for such small size of HDR and MDR transfers.

In order to better show the difference of the four cases, Figure 5.12 was traced. There, it can be seen small differences between the considered cases, mainly because of the different information size sent.

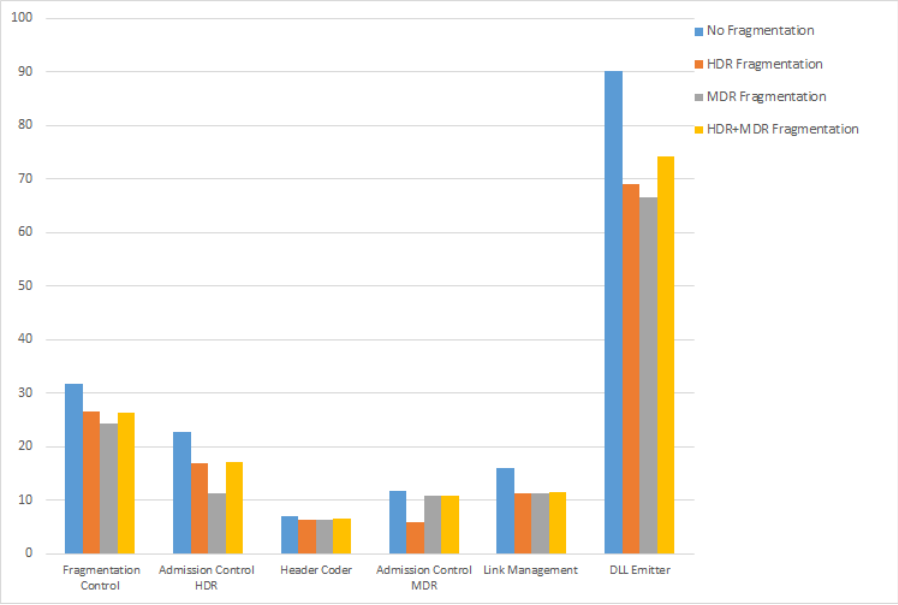

Figure 5.12: Results of the Fragmentation Cases

5.4.5 Time elapsed with different frame sizes

In this section it will be shown the DLL processing functions impact in the system throughput. Since these processing functions have a similar duration despite the fragmentation or data transfer sizes, increasing the DLL frame from the 208 Bytes to 4096 Bytes reveals no real time change in the required functions. Thus, the 4096 Bytes frame size with no fragmentation will be shown in this section as proof of the throughput improvement that is the result of the small differences in the measured times with the big improvement in data transfer size.

Sample number	Fragment Control	Admission Control HDR	Header Coder	Admission Control MDR	Link Management	DLL Emitter
#1	892	861	179	291	495	**3055**
#2	881	873	179	291	489	**3078**
#3	881	822	179	291	501	**3065**
#4	880	855	179	291	492	**3071**
#5	882	861	179	291	495	**3054**
Average:	884	855	179	291	495	**3065**
Time (μs):	17,68	17,1	3,58	5,82	9,9	**61,3**

Table 5.18: Timer samples of DLL Emitter with no fragmentation FrameSize=4096 Bytes

Sample number	Link Management	Header Decoder	Defragmentation Control	Higher Layer Control	DLL Receiver
#1	504	171	122	1008	**2206**
#2	494	171	122	1015	**2204**
#3	506	177	122	993	**2209**
#4	500	171	122	1001	**2203**
#5	503	171	122	1006	**2206**
Average:	502	173	122	1005	**2206**
Time (μs):	10,04	3,46	2,44	20,1	**44,12**

Table 5.19: Timer samples of DLL Receiver with no fragmentation FrameSize=4096Bytes

Given the mentioned timer frequency of 50MHz that corresponds to $20ns$ of sampling period, the last row of 5.18 and 5.19 can be calculated to later be used in the throughput equations. The Emitter throughput result is given in Equation 5.12, where Receiver throughput result is shown in 5.13.

$$Emitter\,Time = Emitter\,Measured\,Samples \times Sampling\,Period = 61,3\mu s \tag{5.10}$$

$$Receiver\,Time = Receiver\,Measured\,Samples \times Sampling\,Period = 44,12\mu s \tag{5.11}$$

$$Throughput\,Emitter = \frac{Frame\,Size\,(4088Bytes)}{Emitter\,Time} = 533,51Mbits/s \tag{5.12}$$

$$Throughput\,Receiver = \frac{Frame\,Size\,(4088Bytes)}{Receiver\,Time} = 741,25Mbits/s \tag{5.13}$$

It is clear that the DLL Emitter and the DLL Receiver have a tremendous throughput increase with the frame size increase. Thus, it is clear that the responsible function for slowing the proposed DLL design is mainly the Fragmentation Control. It is important to remember that Fragmentation Control is responsible for HDR and MDR transfer size control, fragmentation control and input buffer requests control.

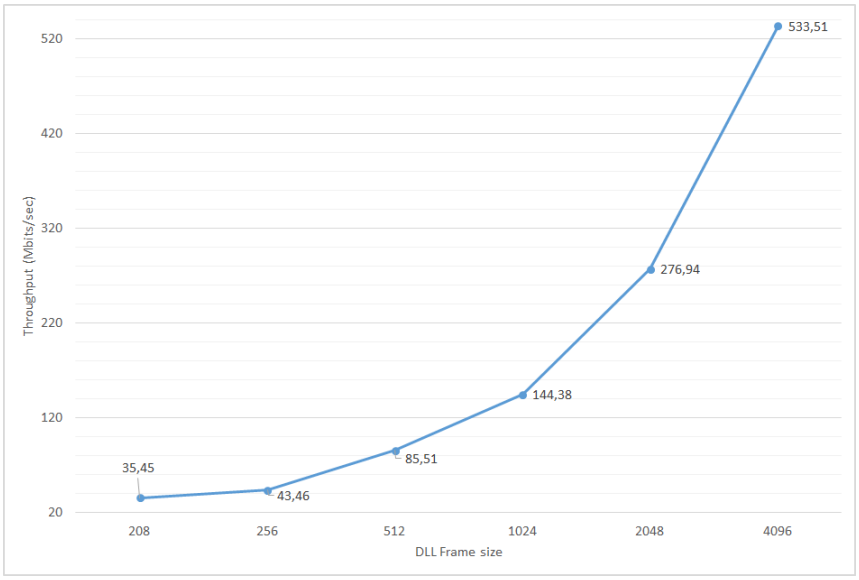

Figure 5.13: DLL Emitter Different frame sizes comparison

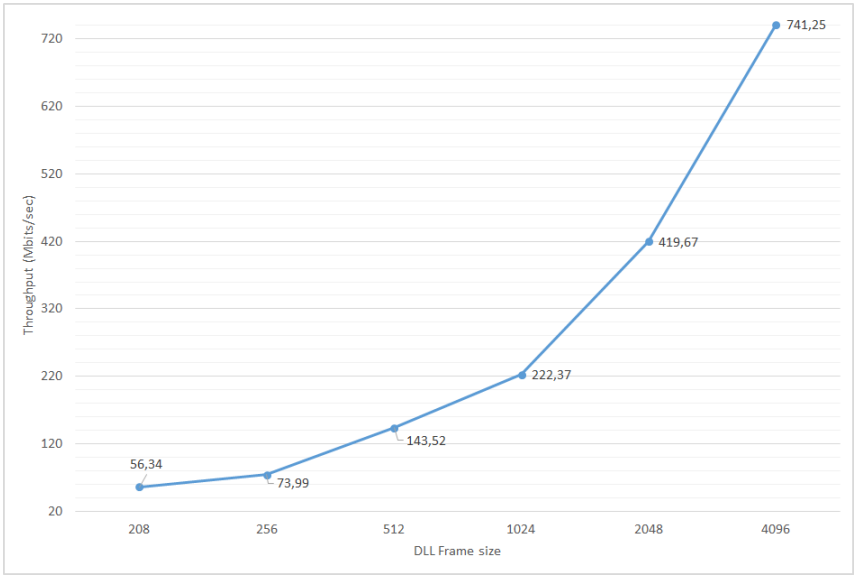

Figure 5.14: DLL Receiver Different frame sizes comparison

With the throughput results obtained with the frame size of 4096 Bytes, it was clear that further study of frame size impact in the system throughput was required. Therefore, frame size was varied between 208 Bytes and 4096 Bytes (maximum addressable frame size due to the header parameters) with values that would not pose addresses alignment issues. These values were 256, 512, 1024 and 4096 Bytes, and their expected impact can be seen in Figures 5.13 and 5.14. Even though these results encourage frame size increase, they were merely tests to show the potential when big transfers are required. All things considered, 208 Bytes frame size present a better choice for the current VLCLighting project demands. It is the best frame size regarding that MPEG-TS frames of 188 Bytes would not completely fill the 4096 Bytes DLL frame size, and the high throughput would be proven to be not worth it.

5.5 Final result analysis

In the previous section 5.4, the DLL function time measurements and their overall analysis on this work was conducted. According to the objectives in designing an efficient DLL with high throughput, the previous shown results were proven to be according to the expected. It is important to notice that the modulation in VLCLighting project, according to the paper [51], is able to provide $24Mb/s$, value that according with expected FEC techniques and DLL efficiencies lowers $21,67\%$, leaving a system throughput of $18,82Mb/s$. So, it is clear that the current DLL is capable of achieving the necessary throughput with the 208 Bytes frame size (35,54 Mbits/sec) and stills leaves room if further improvements in the performance of the remaining blocks occurs.

$$24Mbit/s \times 78,43\% = 18,82Mbits/s < DLL\,Throughtput\,(35,54Mbits/s) \tag{5.14}$$

Chapter 6

Conclusions

This work presented and explored the design of a DLL for VLC broadcast systems with special focus on the VLCLighting project where this work was immersed.

In order to provide the video broadcast (HDR) together with generic data broadcast (MDR), a study of efficiencies trade-off with adjacent blocks in the system architecture was required. This study concluded that the best choice for the proposed system were fixed frames with 208 Bytes of size. This frame size definition together with FEC technique choice of RS(255,213) represents a total system efficiency of $78,43\%$ and was proven to be the best choice regarding VLCLighting project constraints.

After header and payload size establishment, embedded system design was performed in XPS. In this embedded design, a CDMA and Timer IP cores were chosen to fullfill DLL goals. It was also identified that the XPS generated reports show the usage of a small percentage of the available LUTs (7%) of the Spartan 605.

Then, after proper embedded system design and code development, a series of tests were performed to show the DLL capabilities. With these tests, it was left clear that the throughput of the proposed work was limited by the DLL processing functions which are required to handle the fragmentation and data transfer calculations. It was also observed that in this particular FPGA model, the best data transfer size was 64 bits, due to MIG clock frequency limitations. This 64 bits data size choice shows a $8\mu s$ CDMA time to transfer a 208 Bytes. Before measuring implemented functions time, a study of the software profiling and timer limitations was performed. This study concluded that profiling technique was highly unreliable, so Timer was the chosen time measurement. The Timer measurements proved that the proposed DLL could achieve a throughput of $35,45 Mbits/sec$. Since Modulation results show a throughput of $24 Mbits/sec$ and the system efficiency with DLL and FEC usage is imposed as $78,43\%$ (for every 200 Bytes of payload a 255 Bytes need to be sent), the proposed DLL largely suffices the present need of the $18,82 Mbits/sec$.

Even with this satisfactory results, a study of the DLL payload size impact in the DLL throughput was envisioned. This study showed a tremendous DLL performance increase to $533,51 Mbits/sec$ with 4096 frame size and opened the debate for new and improved DLL approaches for future research.

This study opens ground to an improvement in VLC technology data management. During its implementation, it was clear that this DLL could benefit the general VLC broadcast research community with a standard for data packaging and requests management of the OSI Layer3 model.

As for the obtained results, the proposed DLL seems viable to the project where it is inserted. The presented results are satisfactory, because they can cope with current project needs and still leaves many opportunities of improvement that will be presented in the following Section.

6.1 Recommendations and future research

Although the experimental results of the designed DLL are within the required project throughput, it was left clear that further work can be done to improve the proposed layer even further.

The proposed DLL could have a tremendous improvement if the CDMA was configured in Scatter-Gather (SG) mode. For that to happen, the CDMA would also require to be configured with interruptions and the programmed FPGA class code adapted accordingly (SG or Hybrid modes). This future work would largely benefit the transfer data functions that would not require the termination of the DLL processing functions. This would enable transfer requests stacking instead of processor single request configuration each time a transfer is required.

Another identified value added feature in this study was encryption. With data encryption, DLL could ensure the confidentiality of the delivered data to the correct receiver and improve the system protection, especially in this broadcast architecture.

Since the results achieved in Chapter 5 show a huge dependence on the Microblaze processing instructions, the next step to this study improvement would be the DLL programming in a more efficient language like assembly. Therefore, it is with best interest that now that this higher language programming is proved to be a big value added, it could be adapted to a more low level programming and obtain even higher throughputs.

In the performed study, a fixed size DLL frame was identified as the best choice to maintain the lighting functionality of a general VLC architecture. However, in DCO-OFDM or ACO-OFDM modulation cases, that demand would be performed by the modulation and could open a new DLL approach with variable frame size. With this variable frame size, the DLL could obtain a massive throughput improvement, possibly going from the $35,45Mb/s$ to the $533,51Mb/s$ identified. It is very important to regard that this feature would require a new study in the system performance with the remaining blocks. A suggested solution would be FEC codes to take DLL frame, split it in a different number of codewords. This codewords could be treated as the previous codewords size (RS(255,213)), improving system throughput without many changes in the FEC techniques implemented approach. Again, this change would only be justified if value added features would be identified that could benefit from this throughput improvement. In the considered case of HD video broadcast, the MPEG-TS frames will suffice project needs of sending a video frame each time and would not require this increase of frame size. A considered advantage for this scenario would be the sending of more than one MPEG-TS frames or even in cases of video broadcast quality increase.

It is important to remind that the present DLL frame was left with DLL source address field, so further usage of this parameter for localization scenarios or even hand-over warnings could be performed.

DLL frame was also left with reserved bits to implement more required functionalities, like extra requests attendance. With more requests field in the header, DLL could transport two or more requests of the same type and could diminish system packet-jitter time.

Due to code organization, this work presents an easy way to be adapted for different VLC broadcast scenarios. With this in mind, any programmer could for instance adapt the implemented FPGA class to any required FPGA or even run it in any microcontroller (if function calls compatibility with other classes is fulfilled).

Another future work in mind would be the proposed DLL adaptation to bidirectional transmission for VLC systems or heterogeneous systems that combine for example VLC with a return channel in RF.

Bibliography

[1] Ribeiro C., Figueiredo M., Alves L., Duarte L., Rodrigues L., Rodrigues P., "VLCLighting - A Collaborative Research Project on Visible Light Communication." 2015.

[2] P. Ronan, "Electromagnetic Spectrum." http://en.wikipedia.org/wiki/Visible_light_communication, 2007.

[3] W. Reusch, "Visible and Ultraviolet Spectroscopy," 2013.

[4] M. Gancedo, S. Matric, A. Year, V. Light, D. Communications, F. Supervisor, A. Kelly, S. Supervisor, and S. Roy, *Visible light data communications.* PhD thesis, University of Glasgow, 2012.

[5] u. . h. h. . h. Broadband.Gov, title = National Broadband Plan Connecting America.

[6] European Communications Office, "The European table of frequency allocations and applications in the frequency range 8.3KHz to 3000GHz(ECA Table)," tech. rep., 2015.

[7] ANACOM - Autoridade Nacional de Comunicações, "National table of frequency allocation," tech. rep.

[8] Cisco, "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast." http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html, 2015.

[9] "Military Facts of the Great Wall of China." http://greatwall.shanghaifinance.com/great-wallmili1-ta1-ry.php.

[10] "Line of sight communications." http://www.connected-earth.com/Journeys/.

[11] M. Lahanas, "Ancient Greek Communication Methods." http://www.hellenicaworld.com/Greece/Technology/en/Communication.html.

[12] "Hydraulic Telegraph," 2014.

[13] "Heliograph Image." http://upload.wikimedia.org/wikipedia/commons/thumb/9/9b/-Heliograph-2.jpg/220px-Heliograph-2.jpg.

[14] C. H. Sterling, *Military Communications: From Ancient Times to the 21st Century.* 2008.

[15] "Encyclopedia Britannica Heliograph." http://www.britannica.com/EBchecked/topic/-260046/heliograph.

[16] "Bell's Photophone." http://memory.loc.gov/ammem/today/jun03.html, 2010.

[17] "BlueHaze Photophone Image." http://www.bluehaze.com.au/modlight/ModLightBiblio.htm.

[18] "Eugene Polley Flash Matic Telegraph." http://www.telegraph.co.uk/news/obituaries/-9285576/Eugene-Polley.html, 2012.

[19] "Haitz's Law Image." http://organiclighting.com/2012/12/haitz-law-and-the-future-of-led-lighting/.

[20] S. Haruyama, "Japan's Visible Light Communications Consortium and Its Standardization Activities," (Yokohama, Japan), 2007.

[21] "IEEE 802.15 WPAN." http://www.ieee802.org/15/pub/TG7.html, 2009.

[22] "Optics InfoBase OLED VLC." http://www.opticsinfobase.org/oe/abstract.cfm?uri=oe-22-3-2830, 2010.

[23] "500 Megabits/Sec with White LED Light Siemens." http://www.siemens.com/innovation/en/news/2010/500-megabits-second-with-white-led-light.htm, 2010.

[24] "TED VLC." http://www.ted.com/talks/harald_haas_wireless_data_from_every_light_-bulb#t-297663, 2011.

[25] L. Johnson, R. Green, and M. Leeson, "A survey of channel models for underwater optical wireless communication," *2013 2nd International Workshop on Optical Wireless Communications (IWOW)*, pp. 1–5, Oct. 2013.

[26] H. Elgala, R. Mesleh, and H. Haas, "Indoor optical wireless communication: Potential and state-of-the-art," *IEEE Communications Magazine*, vol. 49, pp. 56–62, 2011.

[27] Z. Ghassemlooy, W. Popoola, and S. Rajbhandari, *Optical wireless communications: system and channel modelling with Matlab*. 2012.

[28] D. O'Brien, H. Le Minh, L. Zeng, G. Faulkner, K. Lee, D. Jung, Y. Oh, and E. T. Won, "Indoor Visible Light Communications: challenges and prospects," *Free-Space Laser Communications VIII*, vol. 7091, pp. 1–9, 2008.

[29] Y. Zhao and J. Vongkulbhisal, "Design of visible light communication receiver for on-off keying modulation by adaptive minimum-voltage cancelation," *Engineering Journal*, vol. 17, no. 4, pp. 125–129, 2013.

[30] N. Lourenco, D. Terra, N. Kumar, L. N. Alves, and R. L. Aguiar, "Visible Light Communication System for outdoor applications," in *2012 8th International Symposium on Communication Systems, Networks & Digital Signal Processing (CSNDSP)*, pp. 1–6, 2012.

[31] "EdisonTechCenter." http://www.edisontechcenter.org/LED.html#work.

[32] T. M. Jeffrey A. Hart, Stefanie Ann Lenway, "History of Electroluminescent." http://www.indiana.edu/ hightech/fpd/papers/ELDs.html, 1999.

[33] "World of LEDs." http://www.worldofleds.co.uk/history-of-leds.htm.

[34] "OSRAM History of LEDs,"

[35] "Audi Headlight LEDs." http://www.autoblog.com/2008/05/30/audi-r8-gets-first-full-led-headlamps/.

[36] O. S. Wouter Soer, Ken Vampola, "The road to 250lm/W," 2014.

[37] M. Azadeh, "Fiber Optics Engineering," pp. 157–175, 2009.

[38] F. Laforce, "Low noise optical receiver using Si APD," *SPIE OPTO: Integrated Optoelectronic Devices*, vol. 7212, pp. 0–11, 2009.

[39] R. D. Roberts, "IEEE 802 . 15 . 7 Visible Light Communication : Modulation Schemes and Dimming Support," no. March, pp. 72–82, 2012.

[40] D. V. B. P. Office, "Digital Video Broadcasting (DVB); Interaction channel for Digital Terrestrial Television (RCT) incorporating Multiple Access OFDM DVB Document A064," 2001.

[41] J. R. Cruz, "Error Correction with Reed-Solomon Dr. Doobs." http://www.drdobbs.com/testing/error-correction-with-reed-solomon/240157266, 2013.

[42] G. D. F. Jr., "Concatenated Codes Technical Report 440." http://dspace.mit.edu/bitstream/handle/1721.1/4303/RLE-TR-440-04743368.pdf, 1965.

[43] C. Sciences, "Punctured Convolutional Coding Scheme for Multi-Carrier Multi- Antenna Wireless Systems by Christopher Lamont Taylor,"

[44] "pureLiFi." http://purelifi.com/the-future-of-vlc-modulation-ofdm/, 2013.

[45] D. J. F. Barros, S. K. Wilson, and J. M. Kahn, "Comparison of orthogonal frequency-division multiplexing and pulse-amplitude modulation in indoor optical wireless links," *IEEE Transactions on Communications*, vol. 60, pp. 153–163, 2012.

[46] P. Pascal, "OMEGA ICT Deliverable D4.3 - Optical Wireless MAC Specification," vol. 1, no. 84, pp. 1–84, 2008.

[47] European Telecommunications Standards Institute (ETSI), "Digital video broadcating (DVB); Interaction channel for Digital Terrestrial Television (RCT) Incorporating Multiple Access OFDMA," *ETSI EN 301 958 - DVB-RCT incorporating Multiple Access OFDM*, vol. 1.1.1, pp. 1–164, 2002.

[48] I. Poole, "BER Bit Error Rate Tutorial and Definition." http://www.radio-electronics.com/info/rf-technology-design/ber/bit-error-rate-tutorial-definition.php.

[49] K. Otas, V. J. Pakenas, a. Vaskys, and P. Vaskys, "Investigation of LED light attenuation in fog," *Elektronika ir Elektrotechnika*, vol. 5, no. 5, pp. 47–52, 2012.

[50] J. G. Proakis, *Digital Communications*. Mcgraw-Hill College, third ed., 1994.

[51] C. Ribeiro, M. Figueiredo, and L. N. Alves, "A Real-Time Platform for Collaborative Research on Visible Light Communication,"

[52] D. P. Office, "Generic Stream Encapsulation."

[53] Tektronix, "A Guide to MPEG Fundamentals and Protocol Analysis." http://www.img.lx.it.pt/ fp/cav/Additional_material/MPEG2_overview.pdf.

[54] Xilinx, "Xilinx FPGA Getting started." http://www.xilinx.com/fpga/index.htm.

[55] P. Clarke, "EETimes Xilinx Spartan-6." http://www.eetimes.com/document.asp?doc_-id=1311532, 2099.

[56] E. D. Kit, "MicroBlaze Processor Reference Guide," *Development*, vol. 081, pp. 1–210, 2006.

[57] E. D. Kit, "MicroBlaze Processor Reference Guide," *Development*, vol. 081, pp. 1–210, 2006.

[58] Xilinx, "UG761 AXI Reference Guide," vol. 761, p. 82, 2011.

[59] U. G. July, "SP605 Hardware," vol. 526, pp. 1–70, 2011.

[60] P. Specification, "LogiCORE IP MicroBlaze," pp. 1–37, 2012.

[61] Xilinx, "Xilinx ISE Design Suite." http://www.xilinx.com/products/design-tools/ise-design-suite.html, 2013.

[62] Xillybus, "FPGA coprocessing for C/C++ programmers (part I)." http://xillybus.com/tutorials/vivado-hls-c-fpga-howto-1.

[63] H.-o. Guide, "EDK Concepts, Tools, and Techniques," vol. 683, pp. 1–68, 2011.

[64] I. Xilinx, "Embedded System Tools Reference Manual," vol. 111, 2013.

[65] Xilinx, "XPS Project Files." http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_6/platform_studio/ps_r_gst_project_files.htm.

[66] Xilinx, "OS and Libraries Document Collection," vol. 643, 2014.

[67] Xilinx, "Software workflow in SDK." http://www.xilinx.com/support/documentation/-sw_manuals/xilinx12_4/SDK_Doc/index.html.

[68] Xilinx, "Xilinx SDK Help Contents," 2013.

[69] U. G. April, "A Guide to Profiling in EDK," vol. 448, pp. 1–16, 2012.

[70] B. D. Technology, "High-Level Synthesis Tools for Xilinx FPGAs," *Design*, pp. 1–10, 2010.

[71] Micron, "Micron DDR3 Datasheet." http://www.datasheets360.com/pdf/-5134601209944409612, 2006.

Appendix A

MatLab code

```
%% QPSK BER over AWGN channel
SNR = 0:0.1:10;
BER_QPSK = berawgn(SNR, 'psk', 4, 'nondiff')
figure()
semilogy(SNR, BER_QPSK, 'LineWidth',2)

hold on
grid on

semilogy(7,BER_QPSK(70+1), '*r', 'LineWidth', 2)

strQPSK = strcat('BER = 7.2e-4 @ 7.0dB');
text(6.5, BER_QPSK(70+1),strQPSK,'HorizontalAlignment','right', ...
    'FontSize', 12, 'Color', 'k', 'FontWeight', 'bold', 'Margin', 1, ...
    'BackgroundColor', 'w')

%semilogy(SNR, berawgn(SNR, 'qam', 4), '*r')
xlabel('SNR','FontSize', 12)
ylabel('BER', 'FontSize', 12)
%title('QPSK performance over AWGN channel', 'FontSize', 12)
legend('QPSK')

%% BER_out vs Code efficiency

N = 255;
K_vector = 1:2:249;
R = K_vector./N;

figure()

SNR = 7;

for i=1:length(K_vector)
K = K_vector(i);

BER_out(i) = bercoding(SNR, 'RS', 'hard', N, K, 'psk', 4, 'nondiff');
end

semilogx(BER_out, R, 'LineWidth', 2)
hold on
```

```
39      grid on
40
41      [minimo,indice]=min(BER_out);
42      semilogx(minimo, R(indice), '*r','LineWidth', 2)
43
44      strRScode = strcat('\rightarrow RS(',num2str(K_vector(indice)),',255); ...
            efficiency=', num2str(R(indice),4));
45      text(minimo*4,R(indice),strRScode,'FontSize', 10, 'Color', 'k', ...
            'FontWeight', 'bold', 'Margin', 3, 'BackgroundColor', 'w');
46
47
48      xlabel('Bit Error Rate @ output', 'FontSize', 12)
49      ylabel('Code efficiency', 'FontSize', 12)
50      %title('Bit Error Rate vs efficiency', 'FontSize', 12)
51      legend(strcat('SNR = ', num2str(SNR), 'dB'))
52
53
54
55      %% Efficiency vs FER
56      clear all
57
58      frame_size = [128 256 512 1024 2048 4096];  % frame size in Bytes
59
60      N = 255;
61      K = 165:2:253;
62
63      k = ceil(log2(N+1));            % Elements dimension
64      M = 2^k;                        % Number of different symbols
65      R = K/N;                        % Code rate
66      t = (N-K)/2;                    % Error correction RS capability
67
68      SNR = 7;
69
70      EbNo = 10.^((SNR)/10);       % Linear SNR values
71
72      %p = (1/N)*(1-(1-qfunc(sqrt(2*R.*EbNo))).^k);
73      %PM = p.*(M-1);                 % Symbol error probability
74
75      for f=1:length(frame_size)
76      for c=1:length(K)
77
78      k = ceil(log2(N+1));            % Elements dimension
79      M = 2^k;                        % Number of different symbols
80      R = K(c)/N;                        % Code rate
81      t = (N-K(c))/2;                    % Error correction RS capability
82
83      s = (1-qfunc(sqrt(2*R.*EbNo))).^k;  % Probability of symbol is correctly ...
            received
84      PM = 1-s;                           % Probability of code-word symbol error
85
86      Pes=0;
87      for i = (t+1):N
88      Pes = Pes + (i*nchoosek(N,i).*(PM.^i).*(1-PM).^(N-i));
89      end
90      Pes(f,c) = Pes./N;
91
92      BLER(f,c) = 1-(1-Pes(f,c)).^K(c);
```

```
93
94      NBLOCKS = ceil(frame_size(f)/K(c));
95
96      FER(f,c) = (1-(1-BLER(f,c)).^NBLOCKS);
97
98      end
99
100     end
101
102     %% Efficiency vs FER plots
103     figure()
104     for i=1:length(K)
105     x(:,i)=(K(i)./N) .* (frame_size./(frame_size+8));
106     semilogy((K(i)./N) .* (frame_size./(frame_size+8)), FER(:,i), '-*')
107     hold on
108     grid on
109
110     end
111
112     hold on
113
114     plot(get(gca,'xlim'), [ 2.3704e-08  2.3704e-08])
115
116     %title('Efficiency vs FER Frame ','FontSize', 12)
117     axis([0.65 1 10e-15 2])
118     ylabel('FER','FontSize', 12)
119     xlabel('Total Efficiency','FontSize', 12)
120
121
122     %% Eff vs FER for framesize=256 for all RS codes
123     figure()
124     for i=1:length(K)
125
126     semilogy((K(i)./N) .* (frame_size(2)./(frame_size(2)+8)), FER(2,i), '-*', ...
            'Linewidth', 1.5)
127
128     hold on
129     grid on
130     end
131
132     plot(get(gca,'xlim'), [ 2.3704e-08  2.3704e-08])
133     semilogy((K(25)./N) .* (frame_size(2)./(frame_size(2)+8)), FER(2,25), ...
            '-rO', 'Linewidth', 2)
134
135     strRScode = strcat('\leftarrow RS(',num2str(K(25)),',255); efficiency=', ...
            num2str((K(25)./N) .* (frame_size(2)./(frame_size(2)+8))));
136     text((K(27)./N) .* (frame_size(2)./(frame_size(2)+8)), ...
            FER(2,25),strRScode,'FontSize', 10, 'Color', 'k', 'FontWeight', ...
            'bold', 'Margin', 1, 'BackgroundColor', 'w');
137
138     hold on
139
140     %title('Efficiency vs FER for Frame Size = 256','FontSize', 12)
141     axis([0.65 1 10e-15 2])
142     ylabel('FER','FontSize', 12)
143     xlabel('Total Efficiency','FontSize', 12)
144
```

```
145     %% Efficiency vs BER for one code
146     figure()
147
148     i=25;
149
150     semilogy((K(i)./N) .* (frame_size./(frame_size+8)), FER(:,i), '-*')
151     hold on
152     grid on
153     semilogy((K(i)./N) .* (frame_size(2)./(frame_size(2)+8)), FER(2,i), ...
            '-rO', 'Linewidth', 2)
154
155     axis([0.785 0.85 3e-9 2e-7])
156     ylabel('FER','FontSize', 12)
157     xlabel('Total Efficiency','FontSize', 12)
158
159
160     %% Auxiliar calculations
161     %% Horas a funcionar sem errros para FER
162     bitrate=24e6; %24Mbits/seg
163     framesize=256;    %em bytes
164     frameporseg=bitrate/(8*framesize);
165     tempo=60*60; %tempo em segundos
166     FER=1/(frameporseg*tempo)
167     %% FER para horas
168     FER=1.95*10^-9;
169     bitrate=24000000; %24Mbits/seg
170     framesize=128;    %em bytes
171     tempo_horas=(1/FER)/(bitrate/(framesize*8))/3600
```

Appendix B

DLL C++ code

```
/************ Source file of project ***************/
#include "dll.h"

dll DLL(20);    //Maximum Requests is 32

void HDR_Insert(unsigned int size);
void MDR_Insert(unsigned int size);
/********************** Main **********************/
int main(void)
{
    /* Enable the instruction cache */
    Xil_ICacheEnable();
    /* Enable the data cache */
    Xil_DCacheEnable();

    /* Functions for test purposes */
    DLL.deleteBuffers();
    DLL.test_mode();
    /* Insert Requests */
    HDR_Insert(64000);
    MDR_Insert(3200);
    HDR_Insert(32);
    MDR_Insert(600);
    MDR_Insert(400);

    /* Enter DLL Class */
    xil_printf("Going to enter DLL Class! \r\n");

    DLL.start();

    /* Exit DLL Class */
    xil_printf("--- Exiting main() --- \r\n");

    /* Disable the data cache */
    Xil_DCacheDisable();
    /* Disable the instruction cache */
    Xil_ICacheDisable();

    return 0;
}
```

```
41 /* Fills HDR Buffer In */
42 void HDR_Insert(unsigned int size){
43
44     u32* hdr_ptr =  DLL.getHDRInputPointer();
45     unsigned int tail = DLL.getHDRInputTail();
46     unsigned int i;
47     unsigned int value = 40000;
48     unsigned int words;
49
50     if(size % 8 != 0)
51         size += (8 - size%8);
52
53     words = size /8 ;
54
55     for(i =0; i<words ; i++)
56         *((hdr_ptr+tail)+i) = value++;
57
58     DLL.HDR_Request(size);
59 }
60
61 /* Fills MDR Buffer In */
62 void MDR_Insert(unsigned int size){
63
64     u32* data_ptr = DLL.getMDRInputPointer();
65     unsigned int tail = DLL.getMDRInputTail();
66     unsigned int i;
67     unsigned int value = 5000;
68     unsigned int words;
69
70     if(size % 8 != 0)
71         size += (8 - size%8);
72
73     words = size /8 ;
74
75     for(i = 0; i < words ; i++)
76         *((mdr_ptr+tail)+i) = value++;
77
78     DLL.MDR_Request(size);
79 }
80 /***************** End of main *****************/
81
82 /****************** fifo.h ***********************/
83 #include <stdint.h>
84
85 /* Defines for abstraction and errors */
86 #define HDR 0
87 #define MDR 1
88 #define SUCCESS 0
89 #define ERROR_TYPE_INVALID -1
90 #define ERROR_REQUEST_FIFO_EMPTY -2
91 #define ERROR_REQUEST_FIFO_FULL -3
92 #define ERROR_UNEXPECTED -4
93
94 #ifndef FIFO_H_
95 #define FIFO_H_
96
97
```

```
98  /* FIFO Structure */
99  struct buffer{
100     unsigned short size;    //size of buffer
101     unsigned short start;   //write and read position
102     unsigned short count;   //number of elements in buffer
103     unsigned int *data;     //array of unsigned short pointers
104 };
105 typedef struct buffer buffer_t;
106
107 class fifo {
108
109 public:
110     fifo();
111     fifo(unsigned short fifo_size);
112     virtual ~fifo();
113
114     /* Public Prototypes */
115     int In(unsigned short type, unsigned int value);
116     int Out(unsigned short type);
117     int Check(unsigned short type);
118     int Change(unsigned short type, unsigned int value);
119     void Delete(unsigned short type);
120     bool isEmpty(unsigned short type);
121     bool isFull(unsigned short type);
122
123 private:
124     /* Private Variables */
125     unsigned short fifosize;
126     /* FIFO Structure Declaration */
127     buffer_t buff_hdr_req;
128     buffer_t buff_mdr_req;
129     /* Private Circular FIFO Prototypes */
130     void init(buffer_t *buffer, unsigned short size);
131     void push(buffer_t *buffer, unsigned int value);
132     void change(buffer_t *buffer, unsigned int value);
133     unsigned int pop(buffer_t *buffer);
134     unsigned int check(buffer_t *buffer);
135
136 };
137 #endif /* FIFO_H_ */
138 /*************** End fifo.h ********************/
139
140 /***************** fifo.cpp *********************/
141 #include "fifo.h"
142
143 /* FIFO Class Parameterized Constructor */
144 fifo::fifo(unsigned short fifo_size = 30) {
145     init(&buff_hdr_req, fifo_size);
146     init(&buff_mdr_req, fifo_size);
147     fifosize = fifo_size;
148 }
149
150 /* FIFO Class Destructor */
151 fifo::~fifo() {
152     Delete(HDR);
153     Delete(MDR);
154 }
```

```
155 /* Add Request to FIFO */
156 int fifo::In(unsigned short type, unsigned int value){
157
158     if(type != HDR && type != MDR)                  // Check if Request is Valid
159         return ERROR_TYPE_INVALID;                 // ERROR: REQUEST TYPE INVALID
160
161     if(type == HDR){                                                    // HDR type
162         if(!isFull(HDR)){                          // Check if FIFO HDR is Full
163             push(&buff_hdr_req, value);            // Push Request Size to FIFO
164             return SUCCESS;
165         }
166         else
167             return ERROR_REQUEST_FIFO_FULL;            // ERROR: BUFFER is Full
168     }
169     else if(type == MDR){                                               // MDR type
170         if(!isFull(MDR)){                          // Check if FIFO MDR is Full
171             push(&buff_mdr_req, value);            // Push Request Size to FIFO
172             return SUCCESS;
173         }
174         else
175             return ERROR_REQUEST_FIFO_FULL;            // ERROR: BUFFER is Full
176     }
177     return ERROR_UNEXPECTED;                       // ERROR: Unexpected error
178
179 }
180
181 //Remove Request from FIFO
182 int fifo::Out(unsigned short type){
183
184     if(type != HDR && type != MDR)                  // Check if Request is Valid
185         return ERROR_TYPE_INVALID;                 // ERROR: REQUEST TYPE INVALID
186
187     if(type == HDR){                                                    // HDR type
188         if(!isEmpty(HDR)){                        // Check if FIFO HDR is Empty
189             return pop(&buff_hdr_req);                // Return Value of Request
190         }
191         else
192             return ERROR_REQUEST_FIFO_EMPTY;           // ERROR: BUFFER is EMPTY
193     }
194     else if(type == MDR){                                               // MDR type
195         if(!isEmpty(MDR)){                        // Check if FIFO MDR is Empty
196             return pop(&buff_mdr_req);                // Return Value of Request
197         }
198         else
199             return ERROR_REQUEST_FIFO_EMPTY;           // ERROR: BUFFER is EMPTY
200     }
201     return ERROR_UNEXPECTED;                       // ERROR: Unexpected error
202
203 }
204
205 //Return Request Value without removing it
206 int fifo::Check(unsigned short type){
207
208     if(type != HDR  && type != MDR)                 // Check if Request is Valid
209         return ERROR_TYPE_INVALID;                 // ERROR: REQUEST TYPE INVALID
210
211     if(type == HDR){
```

```
212         if(!isEmpty(HDR)){                          // Check if FIFO HDR is Empty
213             return check(&buff_hdr_req);               // Return Value of Request
214         }
215         else
216             return ERROR_REQUEST_FIFO_EMPTY;      // ERROR: HDR BUFFER IS EMPTY
217         }
218     else if(type == MDR){
219         if(!isEmpty(MDR)){                          // Check if FIFO MDR is Empty
220             return check(&buff_mdr_req);               // Return Value of Request
221         }
222         else
223             return ERROR_REQUEST_FIFO_EMPTY;      // ERROR: MDR BUFFER IS EMPTY
224     }
225     return ERROR_UNEXPECTED;                          // ERROR: UNEXPECTED ERROR
226 }
227
228 //Change Request Value
229 int fifo::Change(unsigned short type, unsigned int value){
230
231     if(type != HDR  && type != MDR)                 // Check if Request is Valid
232         return ERROR_TYPE_INVALID;                // ERROR: REQUEST TYPE INVALID
233
234     if(type == HDR){
235         if(!isEmpty(HDR)){                          // Check if FIFO HDR is Empty
236             change(&buff_hdr_req, value);           // Change value in FIFO HDR
237             return SUCCESS;
238         }
239         else
240             return ERROR_REQUEST_FIFO_EMPTY;      // ERROR: HDR BUFFER IS EMPTY
241     }
242     else if(type == MDR){
243         if(!isEmpty(MDR)){                          // Check if FIFO MDR is Empty
244             change(&buff_mdr_req, value);           // Change value in FIFO MDR
245             return SUCCESS;
246         }
247         else
248         return ERROR_REQUEST_FIFO_EMPTY;          // ERROR: MDR BUFFER IS EMPTY
249     }
250     return ERROR_UNEXPECTED;                          // ERROR: UNEXPECTED ERROR
251 }
252
253 //Remove all Requests in FIFO for that Type
254 void fifo::Delete(unsigned short type){
255
256     if(type==HDR){
257         buff_hdr_req.start = 0;
258         buff_hdr_req.count = 0;
259         buff_hdr_req.size = 0 ;
260         delete [] buff_hdr_req.data;                          // Clear HDR FIFO
261     }
262     else if(type==MDR){
263         buff_mdr_req.start = 0;
264         buff_mdr_req.count = 0;
265         buff_mdr_req.size = 0 ;
266         delete [] buff_mdr_req.data;                          // Clear MDR FIFO
267     }
268 }
```

```
269 //Check if Request FIFO is empty
270 bool fifo::isEmpty(unsigned short type){
271 
272     if(type==HDR){                                                  // HDR type
273         return buff_hdr_req.count == 0;
274     }
275     else if(type==MDR){                                             // MDR type
276         return buff_mdr_req.count == 0;
277     }
278     return true;                                                // Unexpected type
279 }
280 
281 //Check if Request FIFO is full
282 bool fifo::isFull(unsigned short type){
283 
284     if(type==HDR){                                                  // HDR type
285         return buff_hdr_req.count == buff_hdr_req.size;
286     }
287     else if(type==MDR){                                             // MDR type
288         return buff_mdr_req.count == buff_mdr_req.size;
289     }
290     return true;                                                // Unexpected type
291 }
292 
293 /***************** Private Functions *****************/
294 
295 //Initialize the Buffer Variables
296 void fifo::init(buffer_t *buffer, unsigned short size) {
297 
298     buffer->size = size;
299     buffer->start = 0;
300     buffer->count = 0;
301     buffer->data = new unsigned int [size];
302 }
303 
304 //Push Value to that FIFO
305 void fifo::push(buffer_t *buffer, unsigned int data) {
306 
307     unsigned short index;
308     index = buffer->start + buffer->count++;
309     if (index >= buffer->size) {
310         index = 0;
311     }
312     buffer->data[index] = data;
313 }
314 
315 //Pop Value from that FIFO
316 unsigned int fifo::pop(buffer_t *buffer) {
317 
318     unsigned int data;
319     data = buffer->data[buffer->start];
320     buffer->start++;
321     buffer->count--;
322     if (buffer->start == buffer->size)
323         buffer->start = 0;
324     return data;
325 }
```

```
326 //Change Value of that FIFO
327 void fifo::change(buffer_t *buffer, unsigned int value) {
328
329     buffer->data[buffer->start]=value;
330 }
331
332 //Check Value of that FIFO
333 unsigned int fifo::check(buffer_t *buffer){
334
335     return (buffer->data[buffer->start]);
336 }
337 /***************** End fifo.cpp ******************/
338
339 /***************** dllframe.h ********************/
340 #include "xparameters.h"
341 #include "xbasic_types.h"
342
343 #define JUNK  (XPAR_MCB_DDR3_S0_AXI_BASEADDR + 0x00637000)
344
345 #ifndef DLLFRAME_H_
346 #define DLLFRAME_H_
347
348 class dllframe {
349
350 protected:
351     dllframe();
352     virtual ~dllframe();
353     /* Frame size = 208 Bytes */
354     static const unsigned short DLL_FRAME_SIZE = 208;
355     /* Required CDMA Transfers of 64bits(8bytes) for frame transfer */
356     static const unsigned short DLL_FRAME_SIZE_BLOCKS = 26;
357     /* Header size = 8 Bytes */
358     static const unsigned short HEADER_SIZE = 8;
359     /* Required size of 32bits(4bytes) for header transfer */
360     static const unsigned short HEADER_SIZE_BLOCKS = 2;
361     /* Payload size calculation = 200 Bytes */
362     static const unsigned short PAYLOAD_SIZE = DLL_FRAME_SIZE - HEADER_SIZE;
363     /* Maximum allowed fragment number */
364     static const unsigned short MAX_FRAGMENT_NUMBER = 31;
365     /* Maximum allowed request size */
366     static const unsigned int MAX_REQUEST_SIZE = (MAX_FRAGMENT_NUMBER * ...
        PAYLOAD_SIZE);
367     /* Maximum HDR size in frame in case of both HDR and MDR Fragmentation */
368     static const unsigned int MAX_HDR_SIZE = 188;
369     /* Maximum MDR size in frame in case of both HDR and MDR Fragmentation */
370     static const unsigned short MAX_MDR_SIZE = PAYLOAD_SIZE - MAX_HDR_SIZE;
371     /* Maximum established PHY layers to serve */
372     static const unsigned short MAX_NUMBER_PHY_LAYERS = 3;
373     /* DLL Protocol Version set to 1 */
374     static const u32 PROTOCOL_VERSION = 0x01;
375
376     friend class dll;
377     u32* tx_e; //DLLFrame TX Emitter
378     u32* rx_e; //DLLFrame RX Emitter
379     u32* tx_r; //DLLFrame TX Receiver
380     u32* rx_r; //DLLFrame RX Receiver
381     u32* junk; //Junk Frame
```

```
382 private:
383     void initjunkframe(void);
384 };
385 #endif /* DLLFRAME_H_ */
386 /*************** End dllframe.h ******************/
387
388 /**************** dllframe.cpp ******************/
389 #include "dllframe.h"
390
391 /*** Regular Frame ***/
392 dllframe::dllframe() {
393
394     //Initialize the Junk Frame
395     initjunkframe();
396 };
397
398 /*** Destructor Function ***/
399 dllframe::~dllframe() {
400 }
401
402 void dllframe::initjunkframe(void){
403
404     junk = (u32*) JUNK;
405     for(unsigned int i=0 ; i< DLL_FRAME_SIZE_BLOCKS ; i++){
406         *(junk+i) = 1431655765;
407     }
408 }
409 /*************** End dllframe.cpp *****************/
410
411 /******************** dll.h ***********************/
412 #include "dllframe.h"
413 #include "fifo.h"
414 #include "fpga.h"
415 #include <stdio.h>
416 #include <stdint.h>
417
418 /*** Constant Definitions ***/
419 #define HDR 0                 // HDR parameter by default is 0
420 #define MDR 1                 // MDR parameter by default is 1
421 #define FRAME false           // CDMA requests transfer
422 #define NO_FRAME true         // CDMA complete frame transfer
423
424 /* ERROR DEFINITIONS */
425 #define ERROR_TYPE_INVALID -1
426 #define ERROR_REQUEST_FIFO_EMPTY -2
427 #define ERROR_REQUEST_FIFO_FULL -3
428 #define ERROR_UNEXPECTED -4
429 #define ERROR_TRANSFER_BOTH -5
430 #define ERROR_TRANSFER_ -6
431 #define ERROR_TRANSFER_DATA -7
432 #define ERROR_CONFIGURATION_CDMA -8
433
434 #ifndef DLL_H_
435 #define DLL_H_
436
437 class dll {
438 public:
```

```
439     dll(unsigned short buff_size);
440     virtual ~dll();
441     void start(void);
442     void test_mode(void);
443     void deleteBuffers(void);
444     void HDR_Request(unsigned int value);
445     void MDR_Request(unsigned int value);
446 
447     /* Public Prototypes for Higher Layer Access to Image and Data Buffers */
448     u32* getHDRInputPointer(void);
449     u32* getMDRInputPointer(void);
450     unsigned int getHDRInputTail(void);
451     unsigned int getMDRInputTail(void);
452     unsigned int getHDRInputHead(void);
453     unsigned int getMDRInputHead(void);
454     u32* getHDROutputPointer(void);
455     u32* getMDROutputPointer(void);
456     unsigned int getHDROutputTail(void);
457     unsigned int getMDROutputTail(void);
458     unsigned int getHDROutputHead(void);
459     unsigned int getMDROutputHead(void);
460 
461     bool HDR_TO_HIGHER_LAYER;
462     bool MDR_TO_HIGHER_LAYER;
463     bool HIGHER_LAYER_HDR_CLEAR;
464     bool HIGHER_LAYER_MDR_CLEAR;
465 
466 protected:
467     /************ Prototypes ************/
468     void dll_check(u32* emitter_hdr, u32* receiver_hdr, unsigned short ...
        hdr_size, u32* emitter_mdr, u32* receiver_mdr, unsigned short mdr_size);
469 
470     /* Emitter */
471     void emitter_init(void);
472     void operations_controller_emitter(void);
473     void fragmentation_control(void);
474     void header_coder(u8* tx_header);
475     int admission_control(u32* tx_payload, u32* rx_payload, unsigned short ...
        transfer_size);
476     int link_management_emitter(u32* tx, u32* rx);
477 
478     /* Receiver */
479     void receiver_init(void);
480     void operations_controller_receiver(void);
481     int link_management_receiver(u32* tx, u32* rx);
482     void header_decoder(u8* rx_header);
483     void defragmentation_control(void);
484     int higher_layer_control(u32* rx, u32* buffer_hdr, u32* buffer_mdr);
485 
486 private:
487     dllframe frame;
488     fifo request;
489     fpga Fpga;
490 
491     /** Structure Declaration DLL Emitter and DLL Receiver Variables **/
492     struct dllblock{
493         bool more_hdr_frag;
```

```
494         bool more_mdr_frag;
495         bool almost_empty;
496         bool almost_full;
497         unsigned short protocol_version;
498         unsigned short source_address;
499         unsigned short hdr_size;
500         unsigned short mdr_size;
501         unsigned short hdr_frag_number;
502         unsigned short mdr_frag_number;
503         unsigned short HDR_LOST;
504         unsigned short MDR_LOST;
505         unsigned short HDR_COUNTER;
506         unsigned short MDR_COUNTER;
507     };
508     dllblock receiver;  /** Variables Declaration DLL Receiver **/
509     dllblock emitter;   /** Variables Declaration DLL Emitter **/
510
511     struct higherlayerbuffer{
512         u32* ptr;
513         unsigned int tail;
514         unsigned int head;
515         unsigned int count;
516     };
517     higherlayerbuffer hdr_in;
518     higherlayerbuffer mdr_in;
519     higherlayerbuffer hdr_out;
520     higherlayerbuffer mdr_out;
521
522     /* Higher Layer Receiver Control */
523     unsigned short received_hdr_size;
524     unsigned short received_mdr_size;
525     bool hdr_fragment_auxiliar;
526     bool mdr_fragment_auxiliar;
527     bool last_hdr_fragment;
528     bool last_mdr_fragment;
529 };
530 #endif /* DLL_H_ */
531 /****************** End dll.h *********************/
532
533 /****************** dll.cpp **********************/
534 #include "dll.h"
535
536 dll::dll(unsigned short buff_size) : request(buff_size){
537
538     if(buff_size>31)
539         buff_size = 31;
540
541     emitter_init();
542     receiver_init();
543     Fpga.init();
544 }
545
546 dll::~dll() {
547 }
548
549 /*** Initialize the Data Link Layer ***/
550 void dll::start(void){
```

```
551
552     while((!request.isEmpty(HDR)) || (!request.isEmpty(MDR))){
553
554         // DLL EMITTER
555         operations_controller_emitter();
556         // DLL RECEIVER
557         operations_controller_receiver();
558         dll_check(hdr_in.ptr, hdr_out.ptr, emitter.hdr_size, mdr_in.ptr, ...
                mdr_out.ptr, emitter.mdr_size);
559     }
560 }
561
562 /*** Delete all data in Buffers ***/
563 void dll::deleteBuffers(void){
564
565     unsigned short Index;
566     //Reset  buffers of emitter and receiver
567     u32* TX_Emitter = (u32*) TX_EMITTER_BUFFER_BASE;
568     u32* RX_Emitter = (u32*) Fpga.RX_EMITTER_BUFFER_BASE;
569     u32* TX_Receiver = (u32*) Fpga.TX_RECEIVER_BUFFER_BASE;
570     u32* RX_Receiver = (u32*) RX_RECEIVER_BUFFER_BASE;
571     for(Index = 0; Index < frame.DLL_FRAME_SIZE_BLOCKS ; Index++){
572         *(TX_Emitter+Index) = 0;
573         *(RX_Emitter+Index) = 0;
574         *(TX_Receiver+Index) = 0;
575         *(RX_Receiver+Index) = 0;
576     }
577 }
578
579 /*** Initialize DLL in Test Mode ***/
580 void dll::test_mode(void){
581
582     /* Change DLL Receiver End Buffer to DLL Emitter Start Buffer */
583     Fpga.change_to_test_mode();
584     frame.tx_r = (u32*) Fpga.TX_RECEIVER_BUFFER_BASE;
585 }
586
587 /*** HDR Request ***/
588 void dll::HDR_Request(unsigned int value){
589
590     unsigned int words;
591     if(value % 8 != 0)
592         value += (8 - value%8);
593     words = value /8 ;
594     hdr_in.tail += words;
595     hdr_in.count += words;
596     request.In(HDR, value);
597 }
598
599 /*** MDR Request ***/
600 void dll::MDR_Request(unsigned int value){
601
602     unsigned int words;
603     if(value % 8 != 0)
604         value += (8 - value%8);
605     words = value /8;
606     mdr_in.tail += words;
```

```
607     mdr_in.count += words;
608     request.In(MDR, value);
609 }
610 
611 /*************** EMITTER ******************/
612 void dll::emitter_init(void){
613 
614     // TX is the origin buffer of DLL Emitter
615     frame.tx_e = (u32*) TX_EMITTER_BUFFER_BASE;
616     // RX is the destination buffer of DLL Emitter
617     frame.rx_e = (u32*) Fpga.RX_EMITTER_BUFFER_BASE;
618     // Buffer with HDR Data
619     hdr_in.ptr = (u32*) REQ_BUFFER_BASE;
620     hdr_in.head = 0;
621     hdr_in.tail = 0;
622     hdr_in.count = 0;
623     // Buffer with MDR Data
624     mdr_in.ptr = (u32*) REQ_DATA_BUFFER_BASE;
625     mdr_in.head = 0;
626     mdr_in.tail = 0;
627     mdr_in.count = 0;
628     emitter.HDR_COUNTER=0;
629     emitter.MDR_COUNTER=0;
630     emitter.more_mdr_frag = false;
631     emitter.source_address = 0;
632     emitter.hdr_size = 0;
633     emitter.mdr_size = 0;
634     emitter.hdr_frag_number = 0;
635     emitter.mdr_frag_number = 0;
636     emitter.HDR_LOST = 0;
637     emitter.MDR_LOST = 0;
638     emitter.almost_empty = true;
639     emitter.almost_full = false;
640 }
641 
642 /*** OPERATIONS CONTROLLER (EMITTER) ***/
643 void dll::operations_controller_emitter(void){
644 
645     unsigned short size;
646     unsigned short word;
647     int Status;
648 
649     Fpga.StartTimer();
650     fragmentation_control();
651     // Transfer HDR Request Payload to Temporary Frame
652     if(emitter.hdr_size != 0){
653         Status=admission_control(hdr_in.ptr+hdr_in.head, ...
                frame.tx_e+frame.HEADER_SIZE_BLOCKS, emitter.hdr_size);
654         if(Status != XST_SUCCESS)
655             emitter.HDR_LOST++;
656         size = emitter.hdr_size;
657         if(size % 8 != 0)
658             size += (8 - size%8);
659         word = size/8;
660         hdr_in.head += word;
661         hdr_in.count -= word;
662     }
```

```
663     // Transfer Header to Temporary Frame
664     header_coder((u8*)frame.tx_e);
665     // Transfer MDR Request Payload to Temporary Frame
666     if(emitter.mdr_size != 0){
667         word = emitter.hdr_size;
668         if(word % 8 != 0)
669             word += (8 - word%8);
670         Status=admission_control(mdr_in.ptr+mdr_in.head, (frame.tx_e + ...
                frame.HEADER_SIZE_BLOCKS + (word/4)), emitter.mdr_size);
671         if(Status != XST_SUCCESS)
672             emitter.MDR_LOST++;
673         size = emitter.mdr_size;
674         if(size % 8 != 0)
675             size += (8 - size%8);
676         word = size/8;
677         mdr_in.head += word;
678         mdr_in.count -= word;
679     }
680
681     Fpga.StopTimer();
682     // Wait while FEC Buffer is Almost Full
683     while(emitter.almost_full == true);
684     Status = link_management_emitter(frame.tx_e, frame.rx_e);
685     // Check if Transfer was successful and if not add fragment lost to ...
            statistics
686     if(Status != XST_SUCCESS){
687         if(emitter.hdr_size != 0)
688             emitter.HDR_LOST++;
689         if(emitter.mdr_size != 0)
690             emitter.MDR_LOST++;
691     }
692     else{
693         if(emitter.hdr_size != 0)
694             emitter.HDR_COUNTER++;
695         if(emitter.mdr_size != 0)
696             emitter.MDR_LOST++;
697     }
698 }
699
700 /*** FRAGMENTATION CONTROL (EMITTER) ***/
701 void dll::fragmentation_control(void){
702
703     unsigned int hdr_request;
704     unsigned int mdr_request;
705
706     /* Variables to check requests size */
707     if(!request.isEmpty(HDR))
708         hdr_request = request.Check(HDR);
709     else
710         hdr_request = 0;
711     if(!request.isEmpty(MDR))
712         mdr_request = request.Check(MDR);
713     else
714         mdr_request=0;
715
716     // Check if HDR requests are bigger than MAX_REQUEST_SIZE = ...
            MAX_REQUESTS * PAYLOAD_SIZE
```

```
717     while((hdr_request > frame.MAX_REQUEST_SIZE) && (!request.isEmpty(HDR))){
718         hdr_request = request.Out(HDR);
719         emitter.HDR_LOST++;
720         hdr_request = request.Check(HDR);;
721     }
722     // Check if MDR requests are bigger than MAX_REQUEST_SIZE = ...
            MAX_REQUESTS * PAYLOAD_SIZE
723     while((mdr_request > frame.MAX_REQUEST_SIZE) && (!request.isEmpty(MDR))){
724         mdr_request = request.Out(MDR);
725         emitter.MDR_LOST++;
726         data_request = request.Check(MDR);;
727     }
728
729     /* RESET counters of fragment number */
730     if(emitter.more_hdr_frag == false)
731         emitter.hdr_frag_number = 0;
732     if(emitter.more_mdr_frag == false)
733         emitter.mdr_frag_number = 0;
734
735     /***  HDR + MDR > PAYLOAD_SIZE ***/
736     if(hdr_request + mdr_request > frame.PAYLOAD_SIZE){
737         unsigned int size_hdr_remaining;
738         unsigned int size_mdr_remaining;
739
740         /***  HDR + MDR > PAYLOAD_SIZE & MDR < 10%  <- Fragments HDR  ***/
741         if(mdr_request < frame.MAX_MDR_SIZE){
742             emitter.more_hdr_frag = true;
743             emitter.more_mdr_frag = false;
744             emitter.hdr_size = (frame.PAYLOAD_SIZE - mdr_request);
745             size_hdr_remaining = hdr_request - emitter.hdr_size;
746             size_mdr_remaining = 0;
747             request.Change(HDR, size_hdr_remaining);
748             if(!request.isEmpty(MDR))
749                 emitter.mdr_size = request.Out(MDR);
750             else
751             emitter.mdr_size = 0;
752             emitter.hdr_frag_number++;
753             emitter.mdr_frag_number = 0;
754         }
755         else {
756
757             /*** HDR + MDR > PAYLOAD_SIZE & MDR > 10%  & HDR > 90% <- ...
                    Fragments HDR and MDR ***/
758             if(hdr_request > frame.MAX_HDR_SIZE){
759                 emitter.more_hdr_frag = true;
760                 emitter.more_mdr_frag = true;
761                 emitter.images_size = frame.MAX_HDR_SIZE;
762                 emitter.data_size = frame.MAX_MDR_SIZE;
763                 size_hdr_remaining = hdr_request - emitter.hdr_size;
764                 size_mdr_remaining = mdr_request - emitter.mdr_size;
765                 request.Change(HDR, size_hdr_remaining);
766                 request.Change(MDR, size_mdr_remaining);
767                 emitter.hdr_frag_number++;
768                 emitter.mdr_frag_number++;
769             }
770
771             /***  HDR + MDR > PAYLOAD_SIZE & MDR > 10%  & HDR < 90% <- ...
```

```
                Fragments MDR ***/
772             else{
773                 emitter.more_hdr_frag = false;
774                 emitter.more_mdr_frag = true;
775                 if(!request.isEmpty(HDR))
776                     emitter.hdr_size = request.Out(HDR);
777                 else
778                     emitter.hdr_size = 0;
779                 emitter.mdr_size = frame.PAYLOAD_SIZE - emitter.hdr_size;
780                 size_mdr_remaining = mdr_request - emitter.mdr_size;
781                 size_hdr_remaining = 0;
782                 request.Change(MDR, size_mdr_remaining);
783                 emitter.mdr_frag_number++;
784                 emitter.hdr_frag_number = 0;
785             }
786         }
787     }
788
789     /***  HDR + MDR < PAYLOAD_SIZE <- NO FRAGMENTATION ***/
790     else {
791         emitter.more_hdr_frag = false;
792         emitter.more_mdr_frag = false;
793
794         if(!request.isEmpty(HDR))
795             emitter.hdr_size = request.Out(HDR);
796         else
797             emitter.hdr_size = 0;
798
799         if(!request.isEmpty(MDR))
800             emitter.mdr_size = request.Out(MDR);
801         else
802             emitter.mdr_size = 0;
803
804         emitter.hdr_frag_number = 0;
805         emitter.mdr_frag_number = 0;
806     }
807 }
808
809 /*** ADMISSION CONTROL (EMITTER) ***/
810 int dll::admission_control(u32* buffer, u32* tx_payload, unsigned short ...
        transfer_size){
811
812     int Status;
813     if(transfer_size % 8 !=0)
814         transfer_size += (8 - transfer_size%8);
815     Status = Fpga.TransferCDMA(buffer, tx_payload, transfer_size, NO_FRAME, 3);
816     if(Status != XST_SUCCESS)
817         return ERROR_CONFIGURATION_CDMA;
818     return XST_SUCCESS;
819 }
820
821 /*** HEADER CODER (EMITTER) ***/
822 void dll::header_coder(u8* tx_header){
823
824     emitter.protocol_version= (unsigned short)frame.PROTOCOL_VERSION;
825     emitter.source_address = 1;
826
```

```
827     // header[0]=protocol_vers(3bits)+more_hdr_frag(1bit)+hdr_frag_numb(4bits)
828     *tx_header = ((emitter.protocol_version & 0x03) << 5) | ...
            ((emitter.more_hdr_frag & 0x01) << 4) | ((emitter.hdr_frag_number & ...
            0x001E) >> 1);
829     // header[1]=hdr_frag_number(1bit)+hdr_size(7bits)
830     *(tx_header+1) = ((emitter.hdr_frag_number & 0x0001) << 7) | ...
            ((emitter.hdr_size & 0x0FE0) >> 5);
831     // header[2]=hdr_size(5bits)+more_mdr_frag(1bit)+mdr_frag_number(2bit)
832     *(tx_header+2) = ((emitter.hdr_size & 0x001F) << 3) | ...
            ((emitter.more_mdr_frag & 0x01) << 2) | ((emitter.mdr_frag_number & ...
            0x0018) >> 3);
833     // header[3]=mdr_frag_number(3bits) + mdr_size(5bits)
834     *(tx_header+3) = ((emitter.mdr_frag_number & 0x0007) << 5) | ...
            ((emitter.mdr_size & 0x0F80) >> 7);
835     // header[4]=mdr_size(7 bits) + source_address(1bit)
836     *(tx_header+4) = ((emitter.mdr_size & 0x007F) << 1) | ...
            ((emitter.source_address & 0x0800) >> 11);
837     // header[5]=source_address(8bits)
838     *(tx_header+5) = ((emitter.source_address & 0x07F8) >> 3);
839     // header[6] = source_address(3bits)+reserved(5bits)
840     *(tx_header+6) = ((emitter.source_address & 0x0007) << 5 ) & 0xE0;
841     // header[7] = 0
842     *(tx_header+7) = 0x00;
843 }
844
845 /*** LINK MANAGEMENT (EMITTER) ***/
846 int dll::link_management_emitter(u32* tx, u32* rx){
847
848     int Status;
849
850     Status = Fpga.TransferCDMA(tx, rx, frame.DLL_FRAME_SIZE, FRAME, 3);
851     if(Status != XST_SUCCESS)
852         return ERROR_TRANSFER_BOTH;
853     return XST_SUCCESS;
854 }
855 /******************** END EMITTER ************************/
856
857 /********************** RECEIVER ************************/
858 void dll::receiver_init(void){
859
860     // TX is the origin buffer of the DLL Receiver
861     frame.tx_r = (u32*) Fpga.TX_RECEIVER_BUFFER_BASE;
862     // RX is the destination buffer of the DLL Receiver
863     frame.rx_r = (u32*) RX_RECEIVER_BUFFER_BASE;
864     // Exit DLL Buffer with HDR Information
865     hdr_out.ptr = (u32*) RECEIVED_HDR_BUFFER_BASE;
866     hdr_out.head=0;
867     hdr_out.tail=0;
868     hdr_out.count=0;
869     // Exit DLL Buffer with MDR Information
870     mdr_out.ptr =(u32*) RECEIVED_MDR_BUFFER_BASE;
871     mdr_out.head=0;
872     mdr_out.tail=0;
873     mdr_out.count=0;
874     receiver.more_mdr_frag = false;
875     receiver.source_address = 0;
876     receiver.hdr_size = 0;
```

```
877     receiver.mdr_size = 0;
878     receiver.hdr_frag_number = 0;
879     receiver.mdr_frag_number = 0;
880     receiver.HDR_COUNTER=0;
881     receiver.MDR_COUNTER=0;
882     receiver.HDR_LOST = 0;
883     receiver.MDR_LOST = 0;
884     received_hdr_size=0;
885     received_mdr_size=0;
886     hdr_fragment_auxiliar=true;
887     mdr_fragment_auxiliar=true;
888     last_hdr_fragment = false;
889     last_mdr_fragment = false;
890     HDR_TO_HIGHER_LAYER = false;
891     MDR_TO_HIGHER_LAYER = false;
892     HIGHER_LAYER_HDR_CLEAR = false;
893     HIGHER_LAYER_MDR_CLEAR = false;
894     receiver.almost_full = false;
895     receiver.almost_empty = true;
896
897 }
898
899 /*** OPERATIONS CONTROLLER (RECEIVER) ***/
900 void dll::operations_controller_receiver(void){
901
902     int Status;
903     unsigned short size;
904     unsigned short word;
905
906     // DLL buffer Almost Empty, DLL must stop
907     while(receiver.almost_empty==true)
908
909
910     Status = link_management_receiver(frame.tx_r,frame.rx_r);
911     if(Status!=ERROR_TRANSFER_BOTH){
912
913         header_decoder((u8*)frame.rx_r);
914         defragmentation_control();
915         higher_layer_control((frame.rx_r+frame.HEADER_SIZE_BLOCKS), ...
                (hdr_out.ptr+hdr_out.head), (mdr_out.ptr+ mdr_out.head));
916
917         size = receiver.hdr_size;
918         if(size % 8 != 0)
919             size += (8 - size%8);
920         word = size/8;
921         hdr_out.head += word;
922         hdr_out.count -= word;
923
924         size = receiver.mdr_size;
925         if(size % 8 != 0)
926             size += (8 - size%8);
927         word = size/8;
928         mdr_out.head += word;
929         mdr_out.count -= word;
930
931         if(receiver.hdr_size != 0)
932             receiver.HDR_COUNTER++;
```

```
933             if(receiver.mdr_size != 0)
934                 receiver.MDR_COUNTER++;
935 
936             //HDR Request Complete
937             if(last_hdr_fragment == true)
938                 HDR_TO_HIGHER_LAYER = true;
939             //MDR Request Complete
940             if(last_mdr_fragment == true)
941                 MDR_TO_HIGHER_LAYER = true;
942         }
943         else{
944             /* FRAME IS LOST */
945             if(receiver.hdr_size != 0)
946                 receiver.HDR_LOST++;
947             if(receiver.mdr_size != 0)
948                 receiver.MDR_LOST++;
949         }
950 }
951 
952 /*** LINK MANAGEMENT (RECEIVER) ***/
953 int dll::link_management_receiver(u32* tx, u32* rx){
954 
955     int Status;
956 
957     Status = Fpga.TransferCDMA(tx, rx, frame.DLL_FRAME_SIZE, FRAME, 3);
958     if(Status != XST_SUCCESS)
959         return ERROR_TRANSFER_BOTH;
960     return XST_SUCCESS;
961 }
962 
963 /*** HEADER DECODER (RECEIVER) ***/
964 void dll::header_decoder(u8* rx_header){
965 
966     receiver.protocol_version = ((*rx_header & 0xE0) >> 5);
967     receiver.more_hdr_frag = ((*rx_header & 0x10) >> 4);
968     receiver.hdr_frag_number = ((*rx_header & 0x0F) << 1) | ...
        ((*(rx_header+1) & 0x80) >> 7);
969     receiver.hdr_size = ((*(rx_header+1) & 0x7F) << 5) | ((*(rx_header+2) & ...
        0xF8) >> 3);
970     receiver.more_mdr_frag = ((*(rx_header+2) & 0x04) >> 2);
971     receiver.mdr_frag_number = ((*(rx_header+2) & 0x03) << 5) | ...
        ((*(rx_header+3) & 0xE0) >> 5);
972     receiver.mdr_size = ((*(rx_header+3) & 0x1F) << 7) | ((*(rx_header+4) & ...
        0xFE) >> 1);
973     receiver.source_address = ((*(rx_header+4) & 0x01) << 11) | ...
        (*(rx_header+5) << 3) | ((*(rx_header+6) & 0xE0) >> 5);
974 
975 }
976 
977 /*** DEFFRAGMENTATION CONTROL (RECEIVER) ***/
978 void dll::defragmentation_control(void){
979 
980     /* TOTAL HDR SIZE CALCULATION AND SIGNALING OF LAST FRAGMENT */
981     // No fragmentation and Start of Fragmentation
982     if(hdr_fragment_auxiliar == false)
983         received_hdr_size = receiver.hdr_size;
984     // Last Fragment and still waiting for last fragment
```

```
985     else if(hdr_fragment_auxiliar == true)
986         received_hdr_size += receiver.hdr_size;
987     // No fragmentation and Last Fragment
988     if(receiver.more_hdr_frag == false)
989         last_hdr_fragment = true;
990     // Start of Fragmentation and still waiting for last fragment
991     else if(receiver.more_hdr_frag == true)
992         last_hdr_fragment = false;
993 
994     /* TOTAL DATA SIZE CALCULATION AND SIGNALING OF LAST FRAGMENT */
995     // No fragmentation and Start of Fragmentation
996     if(mdr_fragment_auxiliar == false)
997         received_mdr_size = receiver.mdr_size;
998     // Last Fragment and still waiting for last fragment
999     else if(mdr_fragment_auxiliar == true)
1000        received_mdr_size += receiver.mdr_size;
1001    // No fragmentation and Last Fragment
1002    if(receiver.more_mdr_frag == false)
1003        last_mdr_fragment = true;
1004    // Start of Fragmentation and still waiting for last fragment
1005    else if(receiver.more_mdr_frag == true)
1006        last_mdr_fragment = false;
1007
1008    // Store value of current flag of fragmentation
1009    hdr_fragment_auxiliar = receiver.more_hdr_frag;
1010    mdr_fragment_auxiliar = receiver.more_mdr_frag;
1011 }
1012
1013 /*** DATA MANAGEMENT (RECEIVER) ***/
1014 int dll::higher_layer_control(u32* rx, u32* buffer_hdr, u32* buffer_mdr){
1015
1016     int Status;
1017     bool transfer_hdr = true;
1018     bool transfer_mdr = true;
1019     unsigned short transfer_size;
1020     unsigned short word;
1021
1022     // Align Request to 4bytes=32bits due to CDMA
1023     transfer_size = receiver.hdr_size;
1024     if(transfer_size % 8 !=0)
1025         transfer_size = transfer_size + (8 - transfer_size%8);
1026     if(transfer_size != 0){
1027         Status = Fpga.TransferCDMA(rx, buffer_hdr, transfer_size, NO_FRAME, 3);
1028         if(Status != XST_SUCCESS){
1029             transfer_hdr = false;
1030             xil_printf("ERROR: HDR TRANSFER TO HIGHER LAYER FAILED! ");
1031         }
1032     }
1033
1034     // Align Request to 4bytes=32bits due to CDMA
1035     transfer_size = receiver.mdr_size;
1036     if(transfer_size % 8 !=0)
1037         transfer_size = transfer_size + (8 - transfer_size%8);
1038     if(transfer_size != 0){
1039         //Calculate HDR Block transfer
1040         word = receiver.hdr_size;
1041         if(word % 8 != 0)
```

```
1042                word += (8 - word%8);
1043            Status = Fpga.TransferCDMA(rx+(word/4)), buffer_mdr, transfer_size, ...
                    NO_FRAME, 3);
1044            if(Status != XST_SUCCESS){
1045                transfer_mdr = false;
1046                xil_printf("ERROR: MDR TRANSFER TO HIGHER LAYER FAILED! ");
1047            }
1048        }
1049        if(transfer_hdr == false && transfer_mdr == false)
1050            return ERROR_TRANSFER_BOTH;
1051        else if(transfer_hdr == false)
1052            return ERROR_TRANSFER_HDR;
1053        else if (transfer_mdr == false)
1054            return ERROR_TRANSFER_MDR;
1055        else
1056            return XST_SUCCESS;
1057    }
1058    /********************** END RECEIVER ************************/
1059
1060    /*** TEST FUNCTION : DLL frame check ***/
1061    void dll::dll_check(u32* emitter_hdr, u32* receiver_hdr, unsigned short ...
        hdr_size, u32* emitter_mdr, u32* receiver_mdr, unsigned short mdr_size){
1062
1063        bool error = false;
1064
1065        for(int i=0; i< hdr_size/8; i++){
1066            if(*(emitter_hdr+i) != *(receiver_hdr+i)){
1067                xil_printf("HDR SENT INCORRECTLY! \r\n");
1068                error = true;
1069                break;
1070            }
1071        }
1072        for(int i=0; i< data_size/8; i++){
1073            if(*(emitter_mdr+i) != *(receiver_mdr+i)){
1074                xil_printf("MDR SENT INCORRECTLY! \r\n");
1075                error = true;
1076                break;
1077            }
1078        }
1079
1080        if(error != true)
1081            xil_printf("MDR AND HDR SENT CORRECTLY! \r\n");
1082    }
1083
1084    /*** Public Prototypes for Higher Layer Access to HDR and MDR Buffers ***/
1085    u32* dll::getHDRInputPointer(void){
1086        return hdr_in.ptr;
1087    }
1088    u32* dll::getMDRInputPointer(void){
1089        return mdr_in.ptr;
1090    }
1091    unsigned int dll::getHDRInputTail(void){
1092        return hdr_in.tail;
1093    }
1094    unsigned int dll::getMDRInputTail(void){
1095        return mdr_in.tail;
1096    }
```

```
1097 unsigned int dll::getHDRInputHead(void){
1098     return hdr_in.head;
1099 }
1100 unsigned int dll::getMDRInputHead(void){
1101     return mdr_in.head;
1102 }
1103 u32* dll::getHDROutputPointer(void){
1104     return hdr_out.ptr;
1105 }
1106 u32* dll::getMDROutputPointer(void){
1107     return mdr_out.ptr;
1108 }
1109 unsigned int dll::getHDROutputTail(void){
1110     return hdr_out.tail;
1111 }
1112 unsigned int dll::getMDROutputTail(void){
1113     return mdr_out.tail;
1114 }
1115 unsigned int dll::getHDROutputHead(void){
1116     return hdr_out.head;
1117 }
1118 unsigned int dll::getMDROutputHead(void){
1119     return mdr_out.head;
1120 }
1121 /**************** End dll.cpp *********************/
1122
1123 /****************** fpga.h ************************/
1124 #include "xparameters.h"    //defines addresses of all peripherals, ...
        declares the interrupt service routines, declare STDIN/STDOUT devices
1125 #include "xaxicdma.h"       //access the central DMA
1126 #include "xil_exception.h"  //access exception handlers
1127 #include "xintc.h"          //access interrupt controller
1128 #include "xtmrctr.h"        //to access the timer
1129 #include "xbasic_types.h"
1130 #include "xil_cache.h"
1131 #include "xdebug.h"
1132
1133 /**************** Defines ******************/
1134
1135 #define MEMORY_BASE XPAR_MCB_DDR3_S0_AXI_BASEADDR
1136 /*** DLL EMITTER BUFFER ADDRESSES ***/
1137 #define JUNK_BUFFER_BASE (MEMORY_BASE + 0x00637000)
1138 #define JUNK_BUFFER_HIGH (JUNK_BUFFER_BASE + FRAME_SIZE)
1139 #define TX_EMITTER_BUFFER_BASE (MEMORY_BASE + 0x00635000)
1140 #define TX_EMITTER_BUFFER_HIGH (TX_EMITTER_BUFFER_BASE + FRAME_SIZE)
1141 /*** DLL RECEIVER BUFFER ADDRESSES TX Defined in Constructor***/
1142 #define RX_RECEIVER_BUFFER_BASE  (MEMORY_BASE + 0x00839000)
1143 #define RX_RECEIVER_BUFFER_HIGH  (RX_RECEIVER_BUFFER_BASE + FRAME_SIZE)
1144 /*** DLL EMITTER INFORMATION IN BUFFER ADDRESSES ***/
1145 #define REQ_DATA_BUFFER_BASE (MEMORY_BASE + 0x03000000)
1146 #define REQ_DATA_BUFFER_HIGH (MEMORY_BASE + 0x03500000)
1147 #define REQ__BUFFER_BASE (MEMORY_BASE + 0x04000000)
1148 #define REQ__BUFFER_HIGH XPAR_MCB_DDR3_S0_AXI_HIGHADDR
1149 /*** DLL RECEIVER INFORMATION OUT BUFFER ADDRESSES ***/
1150 #define RECEIVED__BUFFER_BASE (MEMORY_BASE + 0x01000000)
1151 #define RECEIVED__BUFFER_HIGH (MEMORY_BASE + 0x01500000)
1152 #define RECEIVED_DATA_BUFFER_BASE (MEMORY_BASE + 0x02000000)
```

```
1153 #define RECEIVED_DATA_BUFFER_HIGH (MEMORY_BASE + 0x02500000)
1154 /********** Constant Definitions ***********/
1155 #define FRAME_SIZE   0x00D0   // 208Bytes
1156 #define HEADER_SIZE  8        // 8Bytes
1157 #define FRAME true            // Transfer DLL Frame
1158 #define NO_FRAME false        // Transfer Higher Layer requests
1159
1160 /* Device hardware build related constants. */
1161
1162 /* Interrupt constants */
1163 #define INTC                  XIntc
1164 #define INTC_HANDLER          XIntc_InterruptHandler
1165 #define INTC_DEVICE_ID        XPAR_INTC_0_DEVICE_ID
1166 /* Timer constants */
1167 #define TMRCTR_DEVICE_ID      XPAR_TMRCTR_0_DEVICE_ID
1168 #define TMRCTR_INTERRUPT_ID XPAR_INTC_0_TMRCTR_0_VEC_ID
1169 #define TIMER_CNTR_0      0
1170 #define RESET_VALUE   0xFD050F80      //Reset value of timer-0xFD050F80=1sec
1171 /* Central DMA constants */
1172 #define DMA_CTRL_DEVICE_ID   XPAR_AXICDMA_0_DEVICE_ID
1173 #define DMA_CTRL_IRPT_INTR   XPAR_INTC_0_AXICDMA_0_VEC_ID
1174 /* Shared variables used to test the callbacks. */
1175 volatile static int TimerExpired;   /* Inserted static */
1176
1177 #ifndef FPGA_H_
1178 #define FPGA_H_
1179
1180 class fpga {
1181
1182 public:
1183     fpga();           // Constructor FPGA Class
1184     virtual ~fpga(); // Destructor FPGA Class
1185
1186     /************ Public Prototypes ************/
1187     unsigned short init(void);
1188     void change_to_test_mode(void);
1189     static unsigned int TransferCDMA(u32* tx, u32* rx, int Length, bool ...
            is_frame, int Retries);
1190     static void StartTimer(void);
1191     static void StopTimer(void);
1192
1193     u32 GetCounterValue(void);
1194     void ResetCounter(void);
1195
1196     u32 TX_RECEIVER_BUFFER_BASE;
1197     u32 TX_RECEIVER_BUFFER_HIGH;
1198     u32 RX_EMITTER_BUFFER_BASE;
1199     u32 RX_EMITTER_BUFFER_HIGH;
1200
1201 private:
1202     /************ Private Prototypes ************/
1203     /* Central DMA Prototypes */
1204     static unsigned int ConfigTimerCDMA(void);
1205     static void DataTransferCheck(u32* SourceBuffer, u32* DestBuffer, int ...
            Size);
1206
1207     /* Timer Prototypes */
```

```
1208      static int SetupIntrSystem(XIntc* IntcInstancePtr, u8 TmrCtrNumber);
1209      static void TimerCounterHandler(void *CallBackRef, u8 TmrCtrNumber);
1210  };
1211
1212  #endif /* FPGA_H_ */
1213  /***************** End fpga.h *********************/
1214
1215  /****************** fpga.cpp **********************/
1216  #include "fpga.h"
1217
1218  /************** Variable Definitions ***************/
1219  static XIntc InterruptController;  /* The instance to the Int. Controller */
1220  static XTmrCtr TimerCounterInst;   /* The instance to the Timer Counter */
1221  static XAxiCdma AxiCdmaInstance;   /* The instance to the XAxiCdma */
1222
1223  /* Temporary Variables to Save Buffer Pointers */
1224  static u32* SavedRXPtr;
1225  static u32* SavedTXPtr;
1226  static u32* PtrJunk;
1227  static int SavedLength;
1228
1229  fpga::fpga() {
1230      /*** DLL RECEIVER BUFFER ADDRESSES ***/
1231      TX_RECEIVER_BUFFER_BASE = (MEMORY_BASE + 0x00835000);
1232      TX_RECEIVER_BUFFER_HIGH = (TX_RECEIVER_BUFFER_BASE + FRAME_SIZE);
1233      RX_EMITTER_BUFFER_BASE = (MEMORY_BASE + 0x00639000);
1234      RX_EMITTER_BUFFER_HIGH = (RX_EMITTER_BUFFER_BASE + FRAME_SIZE);
1235      PtrJunk = (u32*) JUNK_BUFFER_BASE;
1236      //Initialize Transfer Pointer
1237      SavedRXPtr = (u32*) RX_EMITTER_BUFFER_BASE;
1238
1239  }
1240
1241  fpga::~fpga() {
1242  }
1243
1244  unsigned short fpga::init(void){
1245
1246      unsigned short Status;
1247      Status = ConfigTimerCDMA();
1248      if(Status != XST_SUCCESS){
1249          xil_printf("Configuration of CDMA and Timer Failed! \r\n");
1250          return XST_FAILURE;
1251      }
1252      return XST_SUCCESS;
1253  }
1254
1255  void fpga::change_to_test_mode(void){
1256
1257      TX_RECEIVER_BUFFER_BASE = RX_EMITTER_BUFFER_BASE;
1258      TX_RECEIVER_BUFFER_HIGH = RX_EMITTER_BUFFER_HIGH;
1259  }
1260
1261  unsigned int fpga::TransferCDMA(u32* tx, u32* rx, int Length, bool ...
        is_frame, int Retries){
1262
1263      int Status;
```

```
1264      SavedRXPtr = rx;
1265
1266      /* Check if CDMA is currently doing transfer */
1267      while(XAxiCdma_IsBusy(&AxiCdmaInstance));
1268
1269      /* Flush the SrcBuffer of Header addresses before the DMA transfer */
1270      if(is_frame==FRAME)
1271          Xil_DCacheFlushRange((u32)&tx, 2);
1272
1273      /*** DMA Configuration <- Starts when Length is Configured ***/
1274      while(Retries){
1275          Retries-=1;
1276
1277          Status = XAxiCdma_SimpleTransfer(&AxiCdmaInstance, (u32)tx,(u32)rx, ...
                Length, NULL, NULL);
1278
1279          if(Status == XST_SUCCESS)
1280              break;
1281      }
1282
1283      /* Invalidate the DestBuffer of Header addresses before receiving the ...
            data */
1284      if(is_frame==FRAME)
1285          Xil_DCacheInvalidateRange((u32)&rx, 2);
1286
1287      return XST_SUCCESS;
1288  }
1289
1290  /*** TEST FUNCTION: DataTransferCheck ***/
1291  void fpga::DataTransferCheck(u32* SourceBuffer, u32* DestBuffer, int Size){
1292
1293      bool error_result=false;
1294      u32 Index;
1295
1296      for(Index=0; Index < (u32)Size ; Index++){
1297          if(DestBuffer[Index] != SourceBuffer[Index]){
1298              xil_printf("Checked Transfer Function returned with Error! ...
                    Transfer not Successful! \r\n");
1299              error_result = true;
1300              break;
1301          }
1302      }
1303      if(error_result == false)
1304          xil_printf("Checked Sequence with Success! \r\n");
1305  }
1306
1307  void fpga::StartTimer(void){
1308
1309      xil_printf("Timer Started! \r\n");
1310      XTmrCtr_Start(&TimerCounterInst, TIMER_CNTR_0);
1311  }
1312  void fpga::StopTimer(void){
1313
1314      xil_printf("Timer Stopped! \r\n");
1315      XTmrCtr_Stop(&TimerCounterInst, TIMER_CNTR_0);
1316  }
1317
```

```
1318 u32 fpga::GetCounterValue(void){
1319
1320     return XTmrCtr_ReadReg(TimerCounterInst.BaseAddress, TMRCTR_DEVICE_ID, ...
         XTC_TCR_OFFSET);
1321 }
1322
1323 void fpga::ResetCounter(void){
1324
1325 XTmrCtr_Reset(&TimerCounterInst, TMRCTR_DEVICE_ID);
1326 }
1327
1328 /* Configurations of CDMA and Timer */
1329 unsigned int fpga::ConfigTimerCDMA(void){
1330
1331     XAxiCdma_Config *CDMA_CfgPtr;
1332     int Status;
1333
1334     /* Initialize the timer counter so that it's ready to use, specify the ...
         device ID that is generated in xparameters.h */
1335     Status = XTmrCtr_Initialize(&TimerCounterInst, TMRCTR_DEVICE_ID);
1336     if (Status != XST_SUCCESS) {
1337         return XST_FAILURE;
1338     }
1339
1340     /* Initialize the XAxiCdma device. */
1341     CDMA_CfgPtr = XAxiCdma_LookupConfig(DMA_CTRL_DEVICE_ID);
1342     Status = XAxiCdma_CfgInitialize(&AxiCdmaInstance, CDMA_CfgPtr, ...
         CDMA_CfgPtr->BaseAddress);
1343     if (Status != XST_SUCCESS) {
1344         return XST_FAILURE;
1345     }
1346
1347     /* Connect the timer counter to the interrupt subsystem such that ...
         interrupts can occur. This function is application specific.*/
1348     Status = SetupIntrSystem(&InterruptController, &TimerCounterInst, ...
         TIMER_CNTR_0);
1349     if (Status != XST_SUCCESS) {
1350         return XST_FAILURE;
1351     }
1352
1353     /* Setup the handler for the timer counter that will be called from the
1354      * interrupt context when the timer expires, specify a pointer to the
1355      * timer counter driver instance as the call back reference so the ...
          handler is able to access the instance data */
1356     XTmrCtr_SetHandler(&TimerCounterInst, TimerCounterHandler,(void ...
         *)&TimerCounterInst);
1357
1358     /* Enable the interrupt of the timer counter so interrupts will occur
1359      * and use auto reload mode such that the timer counter will reload
1360      * itself automatically and continue repeatedly, without this option
1361      * it would expire once only */
1362     XTmrCtr_SetOptions(&TimerCounterInst, TIMER_CNTR_0, ...
         XTC_CAPTURE_MODE_OPTION | XTC_AUTO_RELOAD_OPTION);
1363
1364     /* Set a reset value for the timer counter such that it will expire ...
         earlier than letting it roll over from 0, the reset value is loaded ...
         into the timer counter when it is started */
```

```
1365      XTmrCtr_SetResetValue(&TimerCounterInst, TIMER_CNTR_0, RESET_VALUE);
1366
1367      return XST_SUCCESS;
1368  }
1369
1370  int fpga::SetupIntrSystem(XIntc* IntcInstancePtr, XTmrCtr* ...
          TmrCtrInstancePtr, u8 TmrCtrNumber){
1371
1372      int Status;
1373      /* Initialize the interrupt controller driver so that it's ready to ...
            use, specify the device ID that is generated in xparameters.h */
1374      Status = XIntc_Initialize(IntcInstancePtr, INTC_DEVICE_ID);
1375      if (Status != XST_SUCCESS) {
1376          return XST_FAILURE;
1377      }
1378
1379      /* Connect the Timer device driver handler that will be called when an ...
            interrupt for the device occurs, the device driver handler performs ...
            the specific interrupt processing for the device */
1380      Status = XIntc_Connect(IntcInstancePtr, ...
            TMRCTR_INTERRUPT_ID,(XInterruptHandler)XTmrCtr_InterruptHandler, ...
            (void *)TmrCtrInstancePtr);
1381      if (Status != XST_SUCCESS) {
1382          return XST_FAILURE;
1383      }
1384
1385      /* Set Options of Interrupt Controller to Return the one with Highest ...
            Priority */
1386      Status = XIntc_SetOptions(IntcInstancePtr,XIN_SVC_SGL_ISR_OPTION);
1387      if(Status != XST_SUCCESS) {
1388          return XST_FAILURE;
1389      }
1390
1391      /* Start the interrupt controller such that interrupts are enabled for
1392       * all devices that cause interrupts, specific real mode so that
1393       * the timer counter can cause interrupts thru the interrupt controller */
1394      Status = XIntc_Start(IntcInstancePtr, XIN_REAL_MODE);
1395      if (Status != XST_SUCCESS) {
1396          return XST_FAILURE;
1397      }
1398
1399      /* Enable the interrupt for the timer counter */
1400      XIntc_Enable(IntcInstancePtr, TMRCTR_INTERRUPT_ID);
1401      /* Initialize the exception table. */
1402      Xil_ExceptionInit();
1403      /* Register the interrupt controller handler with the exception table. */
1404      Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT, ...
            (Xil_ExceptionHandler)INTC_HANDLER, IntcInstancePtr);
1405      /* Enable non-critical exceptions. */
1406      Xil_ExceptionEnable();
1407
1408      return XST_SUCCESS;
1409  }
1410
1411  /*************** Callback Functions ***************/
1412
1413  void fpga::TimerCounterHandler(void *CallBackRef, u8 TmrCtrNumber){
```

```
1414
1415      xil_printf("Timer Interruption Handler Called! \r\n");
1416      XTmrCtr *InstancePtr = (XTmrCtr *)CallBackRef;
1417      if (XTmrCtr_IsExpired(InstancePtr, TmrCtrNumber)) {
1418
1419          xil_printf("Timer Expired! JUNK FRAME WILL BE SENT! \r\n");
1420
1421          /* Stops CDMA and Timer */
1422          XAxiCdma_Reset(&AxiCdmaInstance);
1423          while(!XAxiCdma_ResetIsDone(&AxiCdmaInstance));
1424
1425          /* Transfer Junk Frame */
1426          int Size = XPAR_AXI_CDMA_0_M_AXI_DATA_WIDTH * ...
                  XPAR_AXI_CDMA_0_M_AXI_MAX_BURST_LEN;
1427          TransferCDMA(PtrJunk, SavedRXPtr, Size, FRAME, 3);
1428
1429          /* Acknowledge the Timer Interruption */
1430          XTmrCtr_Reset(InstancePtr, TmrCtrNumber);
1431          XIntc_Acknowledge(&InterruptController, TMRCTR_INTERRUPT_ID);
1432      }
1433  }
1434
1435  /**************** End fpga.cpp ********************/
```

Estes anexos só estão disponíveis para consulta através do CD-ROM.
Queira por favor dirigir-se ao balcão de atendimento da Biblioteca.

Serviços de Biblioteca, Informação Documental e Museologia
Universidade de Aveiro